
ENCYCLOPEDIA OF MICROCOMPUTERS

EXECUTIVE EDITORS

Allen Kent James G. Williams

UNIVERSITY OF PITTSBURGH
PITTSBURGH, PENNSYLVANIA

ADMINISTRATIVE EDITOR

Rosalind Kent

PITTSBURGH, PENNSYLVANIA

VOLUME 1

*Access Methods to Assembly
Language and Assemblers*

MARCEL DEKKER, INC. • NEW YORK and BASEL

DA
76
.15
E52
1988
v.1
C1

Library of Congress Cataloging in Publication Data

Encyclopedia of Microcomputers.

Includes index.

1. Microcomputers--Dictionaries. I. Kent, Allen.
II. Williams, James G. III. Kent, Rosalind.
QA76.15.E52 1987 004.16'03'21 87-15428
ISBN: 0-8247-2700-2

COPYRIGHT © 1988 by MARCEL DEKKER, INC. All Rights Reserved

Neither this book nor any part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage and retrieval system, without permissions in writing from the publisher.

Marcel Dekker, Inc.
270 Madison Avenue, New York, New York 10016

Current printing (last digit):
10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

CONTENTS OF VOLUME 1

<i>Contributors to Volume 1</i>	v
<i>Preface</i>	vii
<i>In Memoriam</i>	ix
ACCESS METHODS <i>Eben Lee Kent</i>	1
ACCESS TECHNOLOGY, INC. <i>Eloise Coupey</i>	7
AEROSPACE DIGITAL CREWSTATIONS <i>John A. Wise</i>	9
ALGOL <i>A. J. Cole</i>	27
ALPHA MICROSYSTEMS <i>Peggy McCranie</i>	57
ALTOS COMPUTER SYSTEMS <i>Michael M. Sanchez</i>	66
AMERICAN SOCIETY FOR INFORMATION SCIENCE—HISTORY <i>Angela Dierking Redmond</i>	74
AMETEK: HOUSTON INSTRUMENT DIVISION <i>Marty McConnell</i>	95
ANALOG-TO-DIGITAL AND DIGITAL-TO-ANALOG CONVERSION <i>Kenneth Sochats</i>	100
ANALYSIS OF THE ECONOMICS OF GAS AND OIL (EGO) <i>Mukund N. Thapa</i>	104
THE ANALYTICAL HIERARCHY PROCESS AND THE PERSONAL COMPUTER <i>Ernest H. Forman and Mary Ann Selly</i>	115
ANIMATION OF NETWORK-BASED SIMULATION MODELS <i>Don N. Pope and M. G. Stroud</i>	139
ANSI — THE AMERICAN NATIONAL STANDARDS INSTITUTE <i>M. E. L. Jacob and D. L. Rings</i>	150
APL FOR MICROCOMPUTERS <i>Maria Tavera</i>	162
APOLLO COMPUTER, INC. <i>David L. Nelson</i>	215
APPLE COMPUTER, INC. <i>Barbara Krause</i>	218
APPLICATION SOFTWARE, EVALUATION <i>Judy Speedy</i>	222
APPLICATIONS SOFTWARE REVIEWING <i>John A. Cross</i>	236
APPLIED DIGITAL DATA SYSTEMS, INC. <i>Thomas J. Deas</i>	240
ARCHITECTURE OF MICROPROCESSORS <i>Robert C. Stanley</i>	245
ARTIFICIAL INTELLIGENCE <i>Derek Partridge</i>	283
ARTIFICIAL INTELLIGENCE FOR AUTOMOBILE MANUFACTURING <i>Atul Bajpai and Barbara A. Sanders</i>	325

ARTIFICIAL INTELLIGENCE APPLICATIONS IN CHEMISTRY	342
<i>Bruce A. Hohne, Thomas H. Pierce, and Mary A. Bright</i>	
ASCII CODE	369
<i>Lee C. Taddonio</i>	
ASHTON-TATE, INC.	376
<i>Ashton-Tate, Inc.</i>	
ASSEMBLY LANGUAGE AND ASSEMBLERS	383
<i>Roger Flynn</i>	

CONTRIBUTORS TO VOLUME 1

ASHTON-TATE, INC., Torrance, California: *Ashton-Tate, Inc.*

ATUL BAJPAI, Senior Project Engineer, Artificial Intelligence, General Motors, Corporation, Warren, Michigan: *Artificial Intelligence for Automobile Manufacturing*

MARY A. BRIGHT, Scientific Programmer, Computer Application Research, Rohm and Haas Co., Spring House, Pennsylvania: *Artificial Intelligence Applications in Chemistry*

A. J. COLE, Department of Computational Science, University of St. Andrews, Fife, Scotland: *ALGOL*

ELOISE COUPEY, Access Technology, Inc., South Natick, Massachusetts: *Access Technology, Inc.*

JOHN A. CROSS, Associate Professor, Computer Science Department, Indiana University of Pennsylvania, Indiana, Pennsylvania: *Applications Software Reviewing*

THOMAS J. DEAS, Manager, Technical Publications, Applied Data Systems, Inc., Hauppauge, New York: *Applied Digital Data Systems, Inc.*

ROGER R. FLYNN, Associate Professor, Interdisciplinary Department of Information Science, University of Pittsburgh, Pittsburgh, Pennsylvania: *Assembly Language and Assemblers*

ERNEST H. FORMAN, D.Sc., Professor and Director of Information Systems, The George Washington University School of Government and Business Administration, Washington, D.C.: *The Analytical Hierarchy Process and the Personal Computer*

BRUCE A. HOHNE, Ph.D., Senior Scientist, Computer Applications Research, Rohm and Haas Co., Spring House, Pennsylvania: *Artificial Intelligence Applications in Chemistry*

M. E. L. JACOB, Vice President, Library Planning, OCLC Online Computer Library Center, Inc., Dublin, Ohio: *American National Standards Institute (ANSI)*

EBEN LEE KENT, Information Systems Product Manager, Standard Rate & Data Service, Inc., Wilmette, Illinois: *Access Methods*

BARBARA KRAUSE, Apple Computer, Inc., Cupertino, California: *Apple Computer, Inc.*

MARTY McCONNELL, Press Relations Specialist, Houston Instrument Division of AMETEK, Inc.: *AMETEK: Houston Instrument Division*

PEGGY McCranie, Alpha Microsystems, Inc., Irvine, California: *Alpha Microsystems, Inc.*

DAVID L. NELSON, Ph.D., Vice President and Chief Technical Officer, Apollo Computer, Inc., Chelmsford, Massachusetts: *Apollo Computer, Inc.*

DEREK PARTRIDGE, Professor of Computer Science, Department of Computer Science, University of Exeter, Exeter, United Kingdom: *Artificial Intelligence*

THOMAS H. PIERCE, Ph.D., Senior Scientist, Computer Applications Research, Rohm and Haas Co., Spring House, Pennsylvania: *Artificial Intelligence Applications in Chemistry*

D. N. POPE, Ph.D. LTV Aerospace and Defense Company, Vought Aero Products Division, Dallas, Texas: *Animation of Network-Based Simulation Models*

ANGELA DIERKING REDMOND, M.L.S., M.B.A., Library and Data Systems Consultant, Alexandria, Louisiana: *American Society for Information Science—History*

D. L. RINGS, Information Analyst, OCLC Online Computer Library Center, Inc., Dublin, Ohio: *American National Standards Institute (ANSI)*

MICHAEL M. SANCHEZ, Manager, Public Relations, Altos Computer Systems, San Jose, California: *Altos Computer Systems*

BARBARA A. SANDERS, Director, Artificial Intelligence, General Motors Corporation, Warren, Michigan: *Artificial Intelligence for Automobile Manufacturing*

MARY ANN SELLY, Director, Decision Support Software, Inc., McLean, Virginia: *The Analytical Hierarchy Process and the Personal Computer*

KENNETH M. SOCHATS, Lecturer, Department of Information Science, University of Pittsburgh, Pittsburgh, Pennsylvania: *Analog-to-Digital and Digital-to-Analog Conversion*

JUDY SPEEDY, School of Library and Information Science, University of Pittsburgh, Pittsburgh, Pennsylvania: *Application Software, Evaluation*

ROBERT C. STANLEY, Senior Engineer, Systems Engineering, Manufacturing Systems Products, International Business Machines Corporation, Boca Raton, Florida: *Architecture of Microprocessors*

M. G. STROUD, LTV Aerospace and Defense Company, Vought Aero Products Division, Dallas, Texas: *Animation of Network-Based Simulation Models*

LEE C. TADDONIO, Lecturer, School of Library and Information Science, University of Pittsburgh, Pittsburgh, Pennsylvania: *ASCII Code*

MARIA L. TAVERA, Ph.D., Systems Engineer, FSC, International Business Machines, S.A.E., Madrid, Spain: *APL for Microcomputers*

MUKUND N. THAPA, Ph.D., President, Stanford Business Software, Inc., Mountain View, California: *Analysis of the Economics of Gas and Oil (EGO)*

JOHN A. WISE, Ph.D., Senior Engineer, Human Sciences Department, Westinghouse R & D Center, Pittsburgh, Pennsylvania: *Aerospace Digital Crewstations*

PREFACE

Computing hasn't been the same since microcomputers were introduced. But why? After all, a computer, micro or other, is still a computer. So what has changed? It has been the penetration of microcomputers into more and more of the workings of our society—with much more to come—that has raised the consciousness about computing to this pervasive level. We are now at the stage where everyone depends on microcomputers in some way. To help society in this adaptation, the editors wish to create a compendium of basic knowledge in our field. Any individual in our society, regardless of his or her orientation or background, should have access to reference materials that can help provide an understanding of microcomputers and their use. It is our hope that this encyclopedia will serve that purpose.

The intent of the editors is to produce an encyclopedia which will be over 5,000 pages long, containing approximately 500 articles published in some 10 volumes. The articles are to be published in alphabetical sequence.

The publication will cover the broad spectrum of microcomputer knowledge in such a way that the novice will gain ready access to current information and obtain an assessment of the future impact of computers.

The encyclopedia will cover microcomputer technology as well as fields in which computer technology has widespread usage. It is aimed at the needs of microcomputer hardware specialists, programmers, systems analysts, engineers, operations researchers and mathematicians. It is also directed to readers in other disciplines where microcomputers are used extensively to solve problems or process data.

The scope of the encyclopedia is developed by the enumeration of an alphabetical list of topics ranging from the specific to the generic. The space devoted to each topic will be determined by its importance, availability of information and whether it is intrinsically or peripherally concerned with computers. The encyclopedia will examine microcomputer history, development and future in our society, as well as current information on the state of the art and science.

The treatment of subject matter will be scholarly and exhaustive, yet straightforward so that most articles will be comprehensible to the novice and stimulating to the informed specialist as well. It should be recognized, however, that various levels of technical and mathematical knowledge will be necessary to comprehend some of the articles. Some articles in each volume will be basic and elementary, others will be very advanced. However, most of the articles will start at a fundamental level and proceed to the state of the art, however sophisticated.

"Forward" cross-references will be minimized, because we feel that there is nothing so frustrating to the reader as a reference to something which will appear in the future. Some cross-references to previously published articles will appear. An in-depth index to the entire encyclopedia will be created after the last volume is published.

There are many judgements inherent in encyclopedia-making, but only one of these will be used as an example. Is it better to have more specific articles, or more generic ones? In the main, the clustering into generic articles makes it more difficult to find an appropriate author whose expertise embraces an entire range of subject matter, and who is willing to write on the topic. The making of an encyclopedia is an exercise in the

"art of the possible" so the availability of authors tends to direct the approach used, even if inconsistently.

We should remind the readers of this encyclopedia that we are taking seriously the lessons of Diderot-d'Alembert in terms of encyclopedia-making as an exercise in the art of the possible. It is hoped that the work, while in progress and upon completion, is judged by the chief criterion used by the editors: to create a work that is truly useful to those who consult it.

This encyclopedia is a companion to the highly successful *Encyclopedia of Computer Science and Technology*, published in 16 volumes from 1975 to 1981. When it came time to consider preparation of a second edition of that work, it became obvious that many of the new developments were in the area of microcomputers. Thus, the decision was to issue supplements to the encyclopedia, and to launch the new *Encyclopedia of Microcomputers*. But, the distinctions between microcomputers and other computers have become blurred, so that it has been necessary to include a few of the same articles in both encyclopedias. This overlap has been kept to a minimum level.

Allen Kent

James G. Williams

University of Pittsburgh
Pittsburgh, Pennsylvania

IN MEMORIAM



ALBERT G. HOLZMAN
October 28, 1921 – May 1, 1985

ENCYCLOPEDIA OF MICROCOMPUTERS

VOLUME 1

ACCESS METHODS

INTRODUCTION

Access, as used in the computer field, is defined as the means of obtaining and making use of information. The information may be either a series of commands (i.e., program code) or data. For purposes of this article, access will focus on data instead of commands, unless otherwise indicated, because the needs for improved access have developed more due to data requirements than command requirements. In fact, most methods have arisen from needs generated from file and data base management where there has been great need to manipulate large amounts of data. It should be understood that many of the methods described herein apply equally to commands as well as data.

Obtaining information is a basic and necessary activity to all aspects of computing, whether it is storing, retrieving, adding, deleting, transferring, or computing. All activities are dependent on accessing information in the fastest manner possible.

The central processing unit (CPU) processes information one piece at a time. Any information resident in primary storage (such as random access memory, or RAM) can be delivered to the CPU at speeds comparable to the CPU's. However, secondary storage devices, such as tapes, disk drives, hard disks, and optical disks, operate at much slower speeds. By nature, these secondary devices are engineered for storing large quantities of information at a fraction of the cost of primary storage often at the expense of speed of access. Thus, the CPU tends to wait for information from secondary storage devices.

In addition, unlike primary storage, secondary storage devices contain moving parts that require time to start up and locate the necessary information. Typically, the speed of secondary storage devices can be as much as 100 times slower than the CPU's speed. Applications requiring extensive access to and from secondary devices experience decreasing performance without support of appropriate access methods.

Additionally, unlike primary storage, mass storage devices cannot address each information element directly. Initially, this was due to the limited number of locations that a microcomputer was able to address. In the days of 8-bit CPUs, for instance, a CPU could only address a maximum of 64,000 characters at any given time. In comparison, disk drives attached to these machines could store over 350,000 character or almost five and one half times as much information.

Another basic requirement for computers is an ability to process information in a flexible manner. In this fashion, then, not only is a computer able to process information in randomized fashion but it can also integrate data and commands into the same storage area.

There have been many hardware and software solutions to meet these requirements. This article will not spend time describing the various hardware approaches, but instead will focus on the software approaches, which include

- Literal (absolute) addressing
- Physical sequential
- Indexed sequential
- Indexed random
- Inverted
- Direct addressing
- Hashing

There are two common components in evaluating any access method: access efficiency and storage efficiency. Access efficiency is the means of measuring the number of physical accesses to perform a single logical access. Storage efficiency is the amount of actual storage space required to store both the information and the means to access that information.

Most software access approaches, as their names indicate, employ one of two methods: sequential or random. Sequential access is the means of processing information in predefined order, generally from beginning to end; random access is the means of processing information in any order. All primary devices and most secondary devices are capable of either access method. However, some, such as tape, are only capable of sequential access and, therefore, may not be capable of utilizing all methods of access described below.

LITERAL (ABSOLUTE) ADDRESSING

Literal addressing is the exclusive method of access used by the CPU for primary storage devices on microcomputers. Each location in primary storage is uniquely identified and addressed. The CPU can retrieve and process information stored in primary storage simply by identifying the address of the information.

The CPU accesses any part of primary storage in one of three methods: direct (or absolute), indirect, or relative addressing. In all cases, the method of access is random in that commands or data do not have to be in any specified form. However, the CPU will usually access the next address unless instructed otherwise.

With direct addressing, the CPU identifies an absolute address to access the information. Often the CPU will be instructed to process information in sequence from one address to another. However, even with direct addressing, the CPU can be instructed to process information in any location.

Indirect addressing can be thought of as random in approach because the address used by the CPU is stored at a specified location. Thus, the CPU is actually instructed to process data at a specified location, which identifies the next location to be processed. Indirect addressing is a powerful computer concept that permits a tremendous amount of commands or data to be reprocessed or reused without storing the information more than once.

The third approach, relative addressing, permits the CPU to process information at a location that must be calculated. The calculation is often simple (e.g., add five), but, like indirect addressing, relative addressing offers tremendous computing power.

All three methods of literal addressing provide for great access and storage efficiency. Direct addressing requires a single physical access per single logical access. Direct addressing does not require any additional

storage to access the information. Both indirect and relative addressing require two physical accesses per single logical access and some additional storage. However, if employed extensively, indirect and relative addressing can become less efficient for both access and storage.

PHYSICAL SEQUENTIAL ACCESS

With physical sequential access, the information is organized as it is stored. Often the information is in some order. In either case, the CPU must access the information by starting at the beginning of the first item of information and process through each item one by one. This is by far one of the easiest methods of access but also one of the slowest in terms of access efficiency. In the worst case, every item must be "read" before the desired one is accessed. For most applications, the average number of accesses is one-half the size of the number of items of information. In the interim processing, a file may require twice the space—one for the file to be read and one for the file that is written. Physical sequential access is nearly 100% storage efficient in many data processing activities. Updating a physical sequential file that is in order, however, can be time consuming and costly of space.

Physical sequential access does have its place in data processing. Large portions of program code are stored sequentially. Utility programs, such as those reading from storage device or writing to a storage device, take advantage of this method. It is to be noted, however, that in these instances, the information read or written is handled as data regardless of whether the information is data or program commands. Also, as long as the information is stored in some order, it is possible to access the next or previous logical item of information.

INDEXING

The next three methods of access take advantage of indexing techniques to improve access. An index is nothing more than organizing information by a key component (usually a key field) and associating some sort of identifier or pointer with each key component in the index file. The pointer generally assists the computer in locating the desired information. The index is organized in some fashion by the key component instead of organizing all the items of information. This approach can save significant time in the accessing process because the computer needs to examine only a portion of the information. Indexing has proven to be extremely successful in file-based data. In these cases, the files are broken down into records and each record contains fields of common information. Users of indices will identify a field as the key field for indexing. In many access methods, the key field must contain unique entries; in others, some duplication can occur. Indices generally work as well for records of a predetermined size (i.e., fixed length) and those of various sizes (i.e., variable length).

INDEXED SEQUENTIAL METHOD

In the indexed sequential approach, the index's key value is predetermined for every entry. A value must appear in every entry and every entry must

be unique. This approach is most effective if the entries are in some logical sequence. If the values are in order, the index only requires a single key for a group of entries, typically the highest value appearing in the key for the records in the grouping. This organization permits the computer to search the entries within the index sequentially. Upon encountering the first entry in the index equal to or greater than the desired item, the computer is directed by the pointer to the grouping, which is, in turn, sequentially searched.

One of the characteristics of this access method, whenever the items are in order, is the ability to develop indices to previously developed indices. In this way, the computer may be required to spend less time sequentially processing each level of indexing and thus speed up the process of retrieving information. One should keep in mind that microcomputers are 10 times faster reading information in a sequential form than in a random form.

Like sequential access, the indexed sequential approach can access the next or previous item in a group easily. Inserting/deleting items in files using indexed sequential access can be a problem if the file must be kept in order. In terms of adding information, an overflow area can be developed to handle the extra elements. However, as the overflow area increases, the access efficiency decreases. Another approach to insertion is to split logical groupings in half to allow for growth. This approach requires extensive computer maintenance.

In general, access efficiency is improved by using indexed sequential techniques. As the number of entries increases within the index, access efficiency decreases. Developing additional indices generally increases access efficiency. There are two areas that affect storage efficiency. First, as the amount of information referenced by the indices increases, the storage efficiency decreases. Second, as the number of indices increases the storage efficiency decreases.

INDEXED RANDOM METHOD

The indexed random method has components similar to the indexed sequential method, although the information requirements are significantly different. Both contain an index that is used to access the information. However, in indexed random, the approach to the index is handled in a nonsequential or random fashion.

Even though the entries in the index do not need to be in order, keeping entries in some order frequently improves access efficiency. Powerful look-up algorithms, such as binary or b-tree searches, can be employed only if the entries in an index are ordered. Otherwise, a sequential method of accessing the index must be employed. Each entry does not require a unique key in the indexed random method. However, the amount of maintenance increases with the number of duplicates. Due to the randomness of the items of information, there must be an index entry for every record. If the index is in order, it is possible to access next and previous records. Otherwise, such access is not possible.

Access is relatively efficient for indexed random files. This is especially true where the index is in sequence. Indexed random is not that efficient in storage as each item has an entry in the index as well. As the file increases, so does the size of the index. Indexed random can be used for either storage or retrieval.

INVERTED ACCESS

An inverted access provides for multiple indices to randomly based information. Its sole purpose is for retrieval. An inverted index consists of the field name, the key entry, and the pointer. Items can have the same key entry and, if they do, will be grouped together because the indices are sorted by entry. Sorted entries also permit the quickest access.

The access efficiency of inverted files is very similar to indexed random, as long as the index in the indexed random approach is ordered. The storage efficiency for each index is relatively efficient. However, given the nature of inverted files, this increased number of indices tends to take up much space. The inverted approach tends to use direct address techniques to locate the information.

DIRECT ACCESS

Direct access provides for a direct correspondence between the key entry and the physical address of the item. The physical address is derived from the key entry and, thus, tends to be a complete or partial numeric value and is sorted. Key entries in the direct access method must be unique.

The direct access method is extremely efficient because an item can be located directly from the key entry. The direct access method also tends to be quite storage efficient. However, its storage efficiency is dependent on the density of the key entries. As density increases, the storage wasted decreases.

Although the direct access method seems very efficient, its characteristics do not make it very useful to implement in many situations. Few files have a field that contains unique entries for all items as well as a full or partial numeric entry they can function as an address.

HASHING

Hashing is a variation of direct access. No index is created with hashing. Instead of an index, the item desired to be retrieved is actually transformed into the direct access address through a randomizing or hashing routine. Some common forms include division, midsquare, folding, radix transformation, piecewise-linear function, digit analysis, algebraic coding, and first-last length.

Because the access address is calculated, it is quite common for the different entries to generate the same access address. These collisions, or synonyms, as they are called, cannot coexist at the home location and must be handled in one of two ways. In one, the entries are relocated to the next available unoccupied location. Entries handled in this fashion are called foreign entries. In this approach, the computer computes the value, accesses the home value, and searches sequentially until the key entry matches. It is to be noted that this approach becomes unworkable as more entries are added and collide with foreign entries.

Another approach to handling collisions is to provide for an overflow area where they can reside and be joined together through linking mechanisms. Through this linking, it is possible to access the home value and walk through

the other entries fairly quickly to the appropriate item. This approach has some inherent difficulties if many entries collide into one location.

Being random in nature, hashing does not normally permit next or previous item access. Nor does hashing typically permit sequential access to the information. Both deletion and insertion of items is possible with this access method. Both processes are more easily handled by the linking process than by the relocation method.

Hashing works best if the space is allocated ahead of time. Moreover, different hashing routines handle various size files diversely. Thus, users of the hashing form of access must have a good idea of the application in order to prepare the necessary space and appropriate hash routine.

Hashing is extremely effective in applications where there is frequent need for addition/deletion of items but infrequent need for access to the items. Applications containing these sorts of characteristics include hotel and airline reservation systems. In general, the other access techniques handle the reverse requirements, namely infrequent need for change but frequent need for access.

Access efficiency of the hashing method depends on two factors. First is the user's understanding of the application. A better understanding of the application will determine the accurate amount of space and the appropriate hashing routine. A second factor is the number of collisions that may occur. As the number of collisions increases, so does the number of accesses. In comparison with other methods, hashing provides for reasonable storage efficiency. However, the greatest factor in efficiency is the allocation of space.

BIBLIOGRAPHY

- Atre, S., *Data Base: Structured Techniques for Design, Performance, and Management*, John Wiley, New York, 1980.
- Forsythe, A. I., T. A. Keenan, E. I. Organick, and W. Stenberg, *Computer Science, A First Course*, John Wiley, New York, 1975.
- Grillo, John P., and J. D. Robertson, *Data Management Techniques*, William C. Brown, Dubuque, IA, 1981.
- Knecht, Ken, "Sling Some Hash," *80 Micro*, 32, 124-125 (September 1982).
- Knuth, Donald, E., *The Art of Computer Programming. Vol. III: Sorting and Searching*, Addison-Wesley, Reading, MA, 1972.
- Pantone, John, "Turning the Tables," *Microcomputing*, 8(3), 100-103 (March 1984).
- Tonkin, Bruce, "Sequential Files: Data All in a Row," *80 Micro*, 79, 88-91 (August 1986).
- Tonkin, Bruce, "Random Selections," *80 Micro*, 80, 90-93 (September 1986).
- Tonkin, Bruce, "Random File Access: Reaching for the Record," *80 Micro*, 81, 140-142 (October 1986).
- Tremblay, Jean-Paul, and P. G. Sorenson, *An Introduction to Data Structures with Applications*, McGraw-Hill, New York, 1976.

EBEN LEE KENT

ACCESS TECHNOLOGY, INC.

Access Technology, Inc., is the developer of 20/20, an integrated spreadsheet modeling package for microcomputers, minis, and mainframes.

A spreadsheet is, quite simply, a natural extension of a decision maker's paper, pencil, and calculator, only much, much quicker and far more accurate.

Spreadsheets are used by sales managers, bookkeepers, marketing directors, inventory control specialists, corporate planners, and manufacturing supervisors. And a spreadsheet's applications are just as varied as its users—planning, budgeting, scheduling, project management, financial reports, and cash flow analysis are just a few.

Integrated spreadsheets are better, more sophisticated versions of this crucial planning tool. Like 20/20, most integrated programs start with an advanced spreadsheet and then add powerful business planning and analysis tools. This integration of features lets more and more professionals of all levels incorporate spreadsheet modeling into their daily planning.

Access' most recent spreadsheet product is 20/20, an advanced spreadsheet combining graphics, data base management, and project modeling. The 20/20 package's powerful links to other software programs have encouraged many leading computer and software companies to incorporate 20/20 into their office automation systems. Typical of such strategic arrangements are the Quotron Q1000 workstation, Information Builders' FOCCALC, the Onyx Office, and NCA's MAXCIM.

The prevalent corporate need to standardize the company planning performed on multiuser computers is successfully addressed by 20/20. The package was designed to allow personal computer and mainframe users to work together on corporate applications. Entire models—data, text, and formulas—can be moved between a wide variety of micros, minis, and mainframes, regardless of operating system. This enables users to plan in a shared environment, not in isolation.

The 20/20 package is typically used in business situations that require a large spreadsheet, utilizing data stored on other application packages and central data bases, combined with graphics, goal seeking, and project modeling. Spreadsheets may be created to a size of 1,000 rows by 1,000 columns, and a context-sensitive, on-line help file is readily available with a single keystroke.

In addition, command files, which automate routine or complex tasks, enable the creation of work-sheet templates, as well as instruction for their use within one file. It is possible to automate extremely complex tasks by using 20/20's branching capability, so that one command file will invoke another. Command files are created with either the spreadsheet file or a text editor.

The 20/20 package's complete documentation contains thorough tutorial and reference sections. "Learning 20/20" introduces novice users to spreadsheet modeling and quickly gets them up to speed on 20/20. More experienced users will find succinct explanations of 20/20's advanced features.

Training is available from Access Technology, either on-site or in-house. In addition, Access markets a 2-hour training videotape, which provides consistent on-site training at the user's convenience. The video is divided into three sections: (a) using a 20/20 model, with instructions for entering values and printing models; (b) building a 20/20 model, including entering and editing data, labels, and formulas; and (c) advanced topics, with information about graphics, data base query capabilities, and command files.

Access Technology was founded in 1979 by Allen Kluchman, the company's president. Kluchman was the first director of advertising at Digital Equipment Corporation. In 1969 he became the first director of marketing at Data General, where he was instrumental in launching the NOVA minicomputer.

Access Technology is a privately owned corporation, with its company headquarters located beside the Charles River in South Natick, Massachusetts. European operations are managed from London, England.

BIBLIOGRAPHY

- "Access Technology's 20/20," *Attage*, 39 (August 1985).
- "Access Technology, Inc.'s 20/20: Is it 1-2-3 for Unix?" *Computerworld* (January 14, 1985).
- Avant, Harry, "20/20: The 1-2-3 of Unix-Land?" *UNIX/World*, 58-65 (June 1985).
- "Beyond '1-2-3' and 'Framework'," *PC Week* (April 23, 1985).
- Boulanger, Noel J., "20/20 Vision," *Digital Rev.* (December 1984).
- "Financial Planning Software," *PC Week* (February 19, 1985).
- "Searle Sees 20/20, By Access, as Its Software Solution," *PC Week* (March 26, 1985).
- Shannon, Terry C., "Spreadsheets," *DEC Professional Magazine*, (July 7, 1985).
- "20/20: An Integrated Spreadsheet for DG Users," *Data Base Monthly* (June 1985).
- "20/20 Multi-User Spreadsheet," *ICP Bus. Software Rev.* (European ed.), 32 (August 1985).

ELOISE COUPEY

AEROSPACE DIGITAL CREWSTATIONS

INTRODUCTION

The early barnstormers crossed the country earning a living by selling airplane rides. They literally flew their fragile craft by sensing the wind in their face, by listening to the sounds the wind made going through the wires, by feeling the vibrations and other sensations they received through the seat of their pants and, most important, by visually noting the aircraft's attitude with reference to the horizon. For fair weather flying, this was all that was needed for such aircraft. As a matter of fact, little more is required to fly the modern version of these early aircraft, a type of powered hang glider known as the ultralight.

As the need arose to fly in all types of weather, however, it quickly became clear that these subjective sensations were totally inadequate. For pilots flying in the clouds, the "seat of the pants" approach to data collection and display could not provide the key piece of data that all pilots need: the attitude of the aircraft in space. The human system of balance depends so much on the eyes that when they are blocked as a data source by the clouds obscuring the horizon, the rest of the human system of balance becomes confused and provides false data to the brain. With this confusion, the brain will often conclude that the aircraft is in a right bank when it is actually banking left. I once flew for 15 minutes with the sensation of being inverted when, in fact, the aircraft was in normal flight. Without the instruments, I would have rolled the aircraft over, thus satisfying my sense of balance, and flown right into the ground.

The recognition of this inadequacy by these early pilots can be demonstrated easily by the following example. In years past, when an airmail pilot was caught above an overcast with no way to get down except to penetrate the clouds, a common procedure was to enter a spin and actually spin down through the cloud deck, hoping that there would be enough clearance underneath to recover. This rather risky approach was selected by pilots because, in a spin, at least they knew the attitude they would be in when they exited the bottom of the clouds. If they had tried to fly down through the clouds using only seat of the pants data, there would be no way for them to know what their real attitude would be when they came out of the clouds. Obviously, if the mail was to arrive independent of the "rain, sleet, or the dark of night," if the fledgling airlines were to ever be successful, and if the military was ever to make full use of the airplane, new and better ways of acquiring, transmitting, and processing data about the aircraft and its environment would have to be developed.

This article traces the steps in the development of aircraft instrumentation and controls and offers a look at what is currently happening and what can be expected in the near term. Such a review reveals that the effective use of this technology will require the application of human factors engineering if the system performance is ever to approach what the technology promises.

MECHANICAL DEVICES

Angle of Attack Indicator

The angle of attack is the most critical factor directly controlled by the pilot in determining the ability of a wing to develop lift. This angle is a datum that impacts the safety of flight in all types of weather conditions. One early method of determining angle of attack simply used a long rag tied to one of the wires between the wings. The wind stretched the rag out along its direction of flow, and a pilot could look at the angle between it and the wing and determine the wing's angle of attack.

Airspeed can also be used to estimate angle of attack and, of course, is important in determining the time of arrival at a destination. For this reason, airspeed indicators received early attention. Some of these airspeed indicators used ordinary anemeters (wind speed gauges like those one might see on a house roof). Another approach was to put a flat metal plate on the end of a spring and mount it in the airstream in such a way that the faster the aircraft flew, the more the increasing air pressure would push the flat plate back. With the addition of a few reference marks, a simple airspeed display was thus available.

Heading Indicator

Another key piece of flight data involves the direction the aircraft is headed. The instrument that was first installed for this purpose was the magnetic compass. (A version of this instrument is still required in every aircraft today.) However, there are problems with using the standard "wet" or "whiskey" compass (so called because of the alcohol used to dampen the motion of the compass card). Motions, such as acceleration, deceleration, climbing, descending, bumps, and turns, cause errors in the reading. Because an aircraft is quite often involved in one of these motions, different means of providing heading data to the pilot was needed. Another problem with the compass involves the mechanics of the display. The wet compass presents the heading numbers backwards. That is, the headings displayed to the left of the current heading are reached most quickly by turning right, and vice versa.

The gyrocompass takes advantage of the gyroscope's stability in space to present a stable source of directional data. The gyroscope was often spun by directing ram air from the airstream across the serrated edges of a rotor and later by vacuum systems and electric motors. It uses this stability to keep the display properly oriented in space in relation to north and thus can be used to determine the current heading. A second advantage the gyrocompass provided was the ability to build a display that provides a "bird's-eye view" of the aircraft and its heading. The bird's-eye view display presents the headings in the proper orientation in relation to the aircraft (e.g., all headings to the left are displayed on the left) and thus greatly reduces the pilot's mental work load, as well as the chance for an error (Fig. 1).

Attitude Indicator

To provide the pilot with the all important data about the aircraft's attitude in space required considerable effort. The instrument that was first developed—a turn and bank indicator (Fig. 2)—can tell the pilot that the aircraft is turning, the direction, and the magnitude of the bank (to a limited

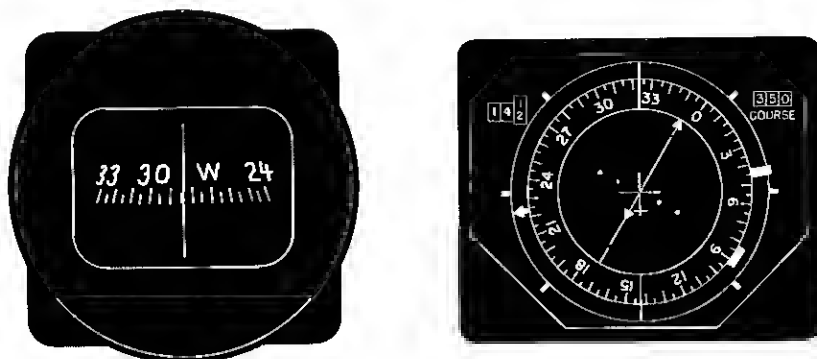


FIGURE 1 Different presentations of heading data.

degree). This indicator uses a gyroscope to provide a stable reference point (similar to the natural horizon) to which the aircraft's attitude can be compared. The turn and bank instrument made flight under instrument conditions possible but still required a great deal of mental work because it did not provide the pilot with a direct indication of pitch (i.e., whether the aircraft nose was high, level, or low). The pilot has to infer that information from the behavior of airspeed and altitude.

The instrument that was to finally provide all of the required attitude information to the pilot was the artificial horizon, or attitude indicator (Fig. 3). It was important because it gave the pilot that important out-the-window visual data that the clouds had taken away, that is, the relation of the aircraft to the horizon. With this instrument, a skilled pilot could easily control the aircraft in areas of poor or no visibility. It was superior to the turn and bank indicator because it provided the pilot with information on both the aircraft's pitch (the location of the nose in relation to the horizon, i.e., nose high or low), as well as bank (the relationship of the wings to the horizon,

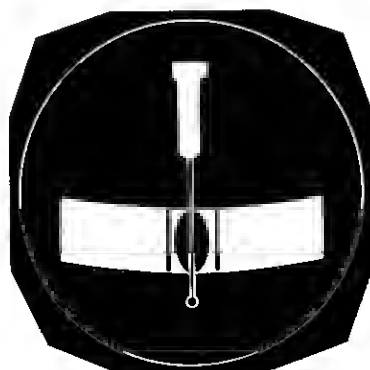


FIGURE 2 Turn and bank indicator.

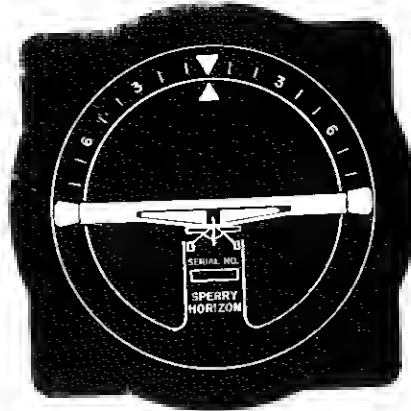


FIGURE 3 Attitude indicator.

i.e., the amount of roll). Early attitude indicators provided only a horizon line as the reference. Because this approach does not distinguish very clearly between an aircraft being nose low inverted and nose high right-side up, later attitude indicators coded the sections of the attitude indicator above and below the horizon (Fig. 4). Typical coding schemes are blue or white for above the horizon, and brown or black for below the horizon.

Location Indicators

The next piece of data required by the pilot is his/her location. This was initially accomplished through automatic direction finding (ADF). ADF is a means of determining and presenting the relative bearing from the aircraft to a transmitting station, that is, it simply points in the direction of the transmitter. However, because it points only to the source, it is difficult to fly a direct course to the station due to the effects of winds.

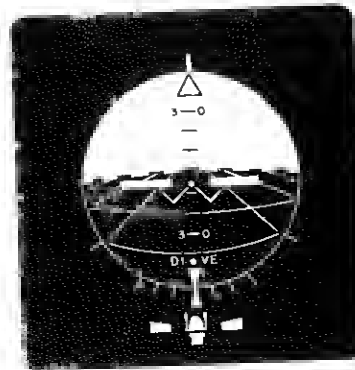


FIGURE 4 Modern attitude indicator.

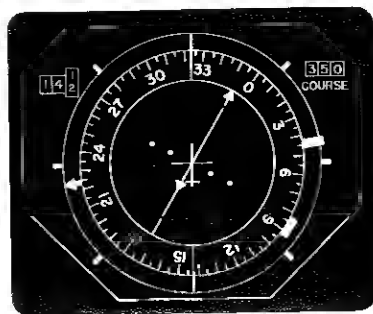


FIGURE 5 HSI display.

Although ADF is still used, these navigational data are now usually acquired and displayed through the use of electronic systems (VOR for civilians and Tacan for the military). These systems translate electronic signals transmitted from a ground station into data on the aircraft's location on one of 360 paths emanating from that ground station. The data are then displayed as the aircraft's lateral deviation from the path selected by the pilot.

Although the VOR display represents a big improvement, it still has one big drawback: Unless the course selected is extremely close to the aircraft's heading, the user has to mentally rotate the display in order to use it. One solution to this problem was the horizontal situation indicator (HSI) (Fig. 5). The HSI display combined the VOR and the gyrocompass on one display. This not only eliminated the need to mentally rotate the VOR display but also effectively combined two related pieces of data into one display, thus further reducing the mental work load required to use it.

ELECTRONIC DEVICES

The first use of electronic displays in aircraft was in the late 1930s, when they were used to display airborne radar. These systems employed cathode ray tubes (CRTs) to present the radar data. They were initially designed to assist military pilots in intercepting enemy aircraft at night, but their use has expanded to include ground mapping, military command and control, and the depiction of certain types of weather to the crew. Other types of sensors have also been added, especially to military aircraft, that use CRTs for their data display technology. They include infrared, television, low light television, and sensors used in smart weapons (e.g., television-based glide bombs).

The head-up display (HUD) was probably the second display to take advantage of electronic media (Fig. 6). Originally used as a gun site on military aircraft, the HUD is mounted on the top of a glare shield right behind the windscreen. Traditionally, it has used a partially silvered mirror (called the combiner glass) to allow the pilot to see both the outside world and the projected aiming device at the same time. It also employs sophisticated optics to focus the projected light as infinity (by collimating the image) so that the

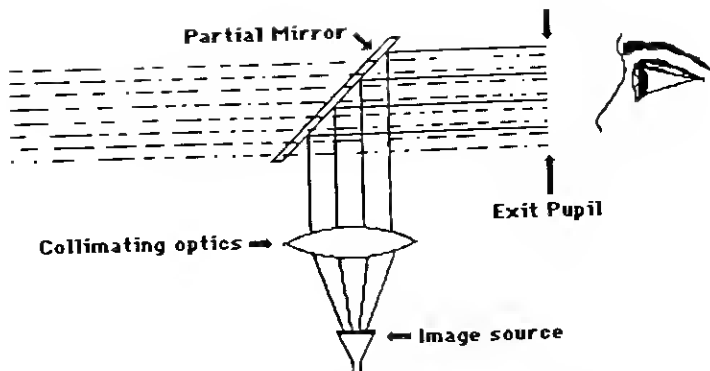


FIGURE 6 Basic optics of a head-up display.

pilot's eyes can see the symbology while simultaneously tracking a target or looking for other traffic. Such optics also make the imagery remain visually fixed in space independent of the pilot's head position. Thus, an aiming device, such as the flight path indicator, will visually appear to remain fixed on the ground no matter where the pilot's head is in the exit pupil.

Current HUDs present all types of flight data, including navigation data, airspeed, altitude, heading, and attitude. Nevertheless, until recently, the HUD was considered a secondary source of data to the pilot. The primary source of flight data was always the dedicated instruments inside the cockpit. With the advent of the Navy/McDonnell-Douglas F/A-18, however, the HUD became the primary flight instrument. It is now intended to be the pilot's primary source of flight information (Fig. 7). The principal reason that the HUD was given this important task is that it allows the pilot to acquire important flight data while simultaneously monitoring the surrounding external environment. For example, during an instrument approach, the HUD allows the pilot to simultaneously fly the aircraft by reference to instruments while looking for the airport through the bad weather.

Mirror-based HUDs do have significant operational limitations. They have a very small area (the exit pupil) where all the symbology can be viewed at the same time. They also have a very small display surface, consequently, the pilot can acquire the flight data in a very small portion of the airspace that must be checked. In addition, the partial mirroring used to reflect the HUD symbology to the pilot's eyes also necessarily blocks some portion of the light from the external environment (and thus information about potential hazards) from reaching the pilot's eyes.

Recent advances in holographic technology have resulted in combiners with high reflectivity in only a limited band of the visual spectrum (Fig. 8). This allows very high brightness of the projected imagery and, at the same time, results in very low attenuation of the outside scene in all but the reflected part of the spectrum. Thus, the pilot can see the outside scenes more easily and yet have a very bright set of flight symbology.

The HUD also has some human factors problems. Pilots report that when they fly through weather while using the HUD as the primary flight instrument, broken clouds racing by the cockpit can be quite distracting. They

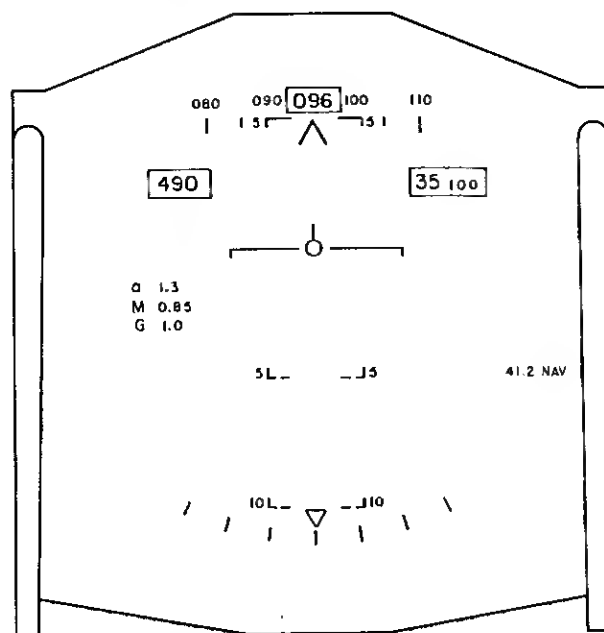


FIGURE 7 HUD imagery.

report that their eyes tend to follow the motions of the clouds and are distracted from their data collection tasks. Pilots have also reported that HUD imagery projected on outside objects (e.g., buildings on the ground) that the pilot knows to be far apart, gives the feeling that his/her eyes must move an extraordinary distance to cross-check the images even though they are physically very close together.

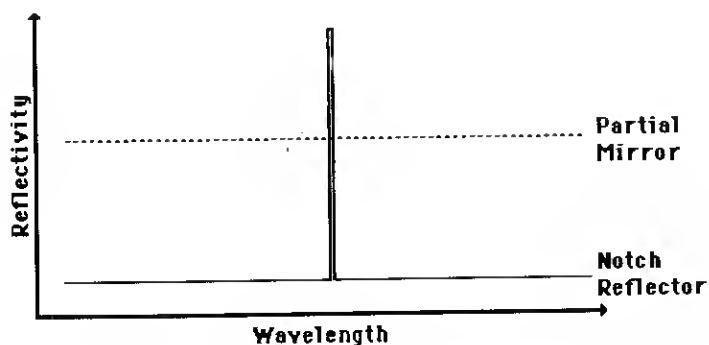


FIGURE 8 Reflectivity characteristics of different types of HUD combines.

Single vs. Mixed Displays

As one can see from the discussion above, the approach to the cockpit design has generally been to design and develop a display so that it serves a single purpose, for example, altitude or airspeed. As a result, most aircraft cockpits today appear very complicated because they have such a large array of single-purpose instruments in the cockpit.

To help the pilot cope with this increasingly complicated array, human factors research has identified the individual displays that are used together. Researchers have found that by combining these individual displays into single displays, they often present a much better picture to the crew. Examples of this integration include the HSI (described earlier), which takes data from a number of related displays (e.g., compass, VOR, ADF) and organizes them into a single whole based on their interrelationship. This approach not only reduces the clutter in the cockpit but also, when done properly, reduces the mental work load required by the pilot to collect and process the needed information.

The reduction in the number of displays needed in a cockpit also has another, rather pragmatic outcome. To enable aircraft to fly faster and use less fuel, designers must reduce the drag the aircraft has to overcome. One of the most crucial factors in drag reduction is the size of the frontal area of the aircraft. Reductions in this area require small cockpits, which can be achieved only by eliminating displays or combining them (Fig. 9).

RATIONALE FOR DIGITAL CREWSTATION

Limitations of Mechanical/Electromechanical Approaches

High reliability of aircraft instrumentation is a necessity for safe flight. Mechanical and electromechanical devices with their many moving parts necessarily have a relatively low mean time between failures. Because there is a great deal of vibration and wide swings of temperature associated with flight,

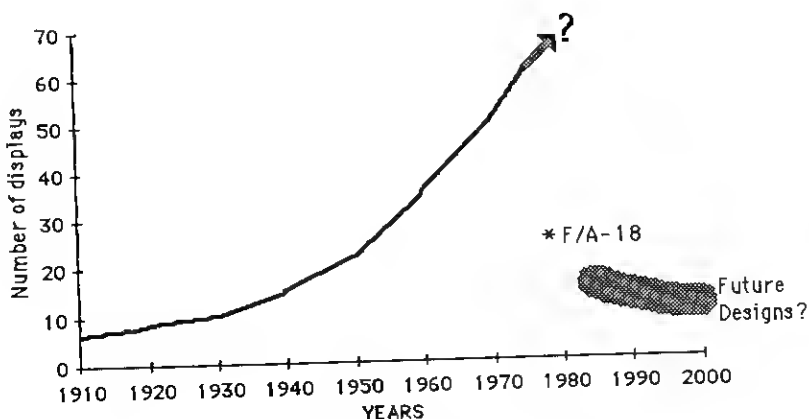


FIGURE 9 Number of displays in fighter-type aircraft.

the limitations of mechanical/electromechanical devices are further amplified. Thus, these devices must be made more rugged, which increases cost and weight. Additionally, a great deal of extra space is needed to provide for redundant systems and redundant displays in the cockpit.

Mechanical- and electromechanical-based displays are extremely limited in the ways that they can present their data because they can be made to work only in certain ways. For example, the integration of a number of different displays into a single integrated version (when such an integration is possible) has typically been limited by the mechanical complexity (and thus lower reliability) that such a device would require. This state of affairs results in data display methods being driven not by the information needs of the pilot but by the limitations of mechanics.

Digital Display Advantages

Digital cockpits bring the advantages of much higher reliability. Digital displays significantly increase mean time between failure times for aircraft display and control systems. Not only does this increase safety but it also increases availability of the aircraft and reduces maintenance costs.

The digital cockpit has also allowed the cockpit designer to begin to create displays that are free of the constraints of mechanical systems and that can be optimally designed to meet the information needs of the pilots. Because of the virtually unlimited data-formatting possibilities that the digital crewstation allows, cockpit designers now find themselves in the position of being able to use human factors engineering guidelines to optimize the presentation of data to the pilot to a degree that has never been possible in the past. In this one regard, the digital crewstation may result in the single biggest change in the work load level of the pilot since the beginning of aviation.

The digital cockpit has had a very significant impact on the number of displays required in the cockpit. The use of a single display surface to present a large number of different display formats creates a significant decrease in the need for cockpit space to be used for dedicated displays. When digital displays are designed properly, they present all the data that a person needs to do a certain task. Thus, data not needed during parts of a flight do not take up valuable space in the cockpit during these parts of the mission.

Additionally, digital components can typically be packaged in much smaller spaces and tend to weigh less than their mechanical or electromechanical counterparts. This makes possible a smaller and lighter aircraft which, in turn, results in improved performance and operating economy.

Digital Control Advantages

Although the previous discussion has addressed only the display problem, the use of digital systems can have an equally significant impact on the design of controls. Solid-state control devices, such as the force transducers, can also bring significant improvement in reliability to control design. The mean time between failures for certain types of solid-state controls can be longer than the expected life of the airframe.

Another advantage of digital controls is that they make it easier for the designer to specify the response characteristics of the control output. For example, the ratio of the movement of the control to the movement of a

cursor on a display (known as the control-display ratio) is a critical factor in the amount of work that a pilot will be required to perform for an exact placement task, for example, placing a cursor over the exact location of a checkpoint. When the cursor moves only slightly for a large amount of control movement, it is easy to place the cursor very precisely, but a relatively long travel time is required if the cursor starts a long distance from the target. On the other hand, a great amount of cursor movement for a small control movement will allow the cursor to reach the general area very quickly but will result in a significant difficulty and delay in getting an accurate placement. With a digital control it is possible to allow both slow (high precision) input and fast (large movement) input on the same control. For example, lighter pressures on a force transducer can produce the high-precision cursor movements, whereas the high pressures can be used for the greater rates movement of the cursor.

When digital devices are used to control the actual flight of the aircraft (known as a fly-by-wire control system), they can provide an even more exciting capability. With the traditional cable (or cable/hydraulic) controls, the effect that a given movement of the control stick has on the attitude of an aircraft varies dramatically with a number of different variables, including speed and altitude. In high-performance fighter aircraft, the control stick must be moved several inches at low speeds to get the same amount of response as a slight pressure will command at supersonic speeds. Digital controls, however, allow control laws to be developed so that the controls either always feel the same or vary only slightly (to provide the pilot with feedback about the performance of the aircraft). And perhaps the most exciting possibility is that the control laws could be designed to meet the individual preference or needs of each pilot. Thus, the intelligent application of digital controls to cockpits should increase reliability and reduce pilot work load.

Recent Examples of Digital Crewstations

Although digital devices have slowly been replacing old analog devices for many years, perhaps the most significant and clean departure in terms of crewstation design occurred in the design of the crewstation of the U.S. Navy's F/A-18 by the McDonnell Aircraft Company. The F/A-18 was the first aircraft to be designed to use all digitally driven displays as the primary source of flight data to the pilot (Fig. 10). It uses three CRTs mounted on the main instrument panel and a CRT-driven HUD as primary displays. It is thus the first operational aircraft to bump the traditional mechanical and electromechanical devices to a backup role. The F/A-18 also utilizes a fly-by-wire control system (although it was not the first operational aircraft to do so.)

As a result of the success of the application of digitally based display technology on the F/A-18, later versions of the General Dynamics F-16, the McDonnell-Douglas F-15E, and the AV-8 have also incorporated CRT displays. Commercial aircraft have also begun to incorporate digitally based displays and controls. The Boeing 757 and 767, the Airbus A310, and some McDonnell-Douglas MD-80 series all have CRT displays. Likewise, the space shuttle has a number of CRT displays for the presentation of data. However, because of its earlier design period and more conservative design philosophy, the shuttle uses the CRT displays only for nonflight control data.

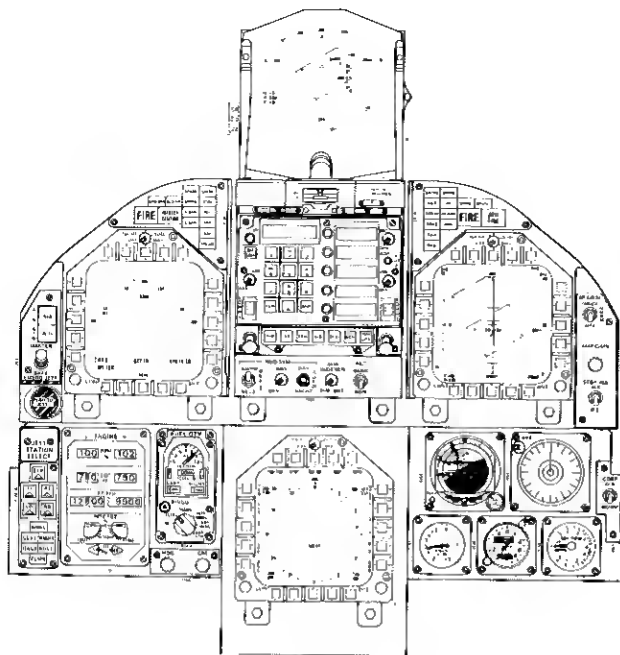


FIGURE 10 F/A-18 cockpit.

The smaller general aviation aircraft have also begun to incorporate digital technology in the area of communications, navigation, and weather radar systems. Examples include the electronic display of radio frequencies and course deviation data, where the electronic display replaces the old electromechanical model through the use of light bars.

At the present time, a significant number of technologies appear to have some potential for aircraft applications. Figure 11 shows those that have received the most attention in recent years. Although it is not appropriate to describe each of them here, it is interesting to look at one of the more sophisticated applications of such technologies. The helmet-mounted display (HMD) utilizes the same general virtual/collimated display techniques as the HUD discussed earlier; however, it uses the visor of the pilot's helmet for the combining glass (Fig. 12). The image source can be any one of a number of different display technologies, for example, CRT or LCD. The advantage of the HMD over the HUD is that it moves with the head, and therefore the data it presents are always available to the pilot no matter where his/her head is pointed.

The HMD can also present sensor data to the pilot. For example, in one application an infrared camera is pointed in the same direction as the pilot's head, and the infrared imagery is then presented to the pilot using the HMD. This system gives the pilot a virtual real-world image that can be used in much the same way a pilot uses the actual image of the real world to control the aircraft in daylight conditions.

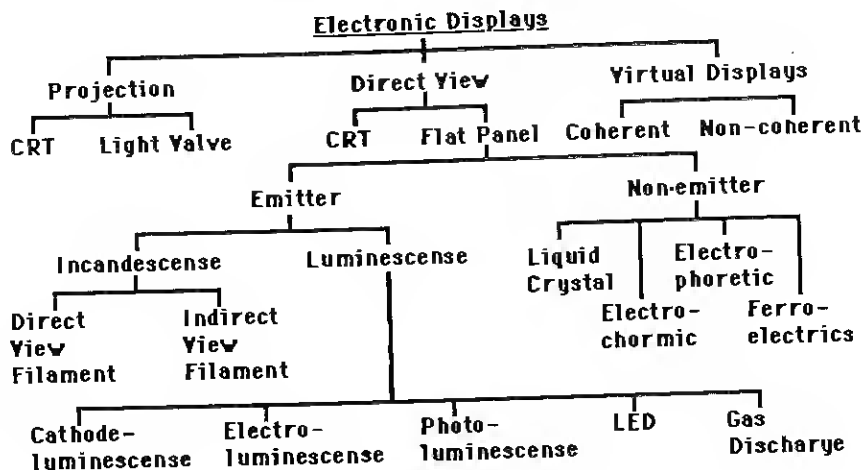


FIGURE 11 Electronic display.

CRITIQUE OF CURRENT APPLICATIONS

Strengths

The initial designs using digital equipment have been unusually successful for a number of reasons. First, there was a great deal of basic research performed in attempts to determine better ways to provide data and to control the aircraft. The military, in particular, tried to determine better ways of

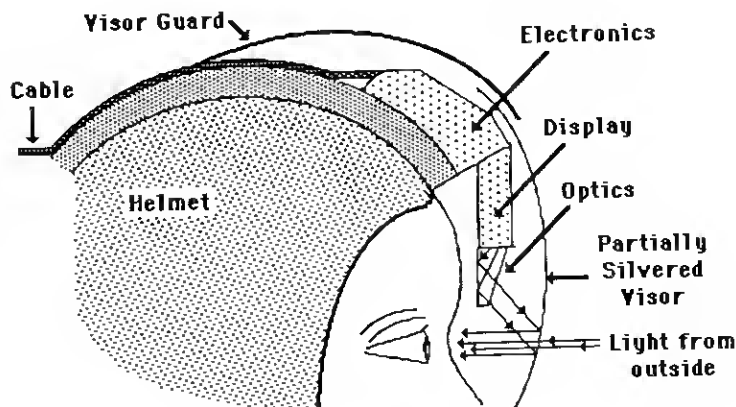


FIGURE 12 HMD optics.

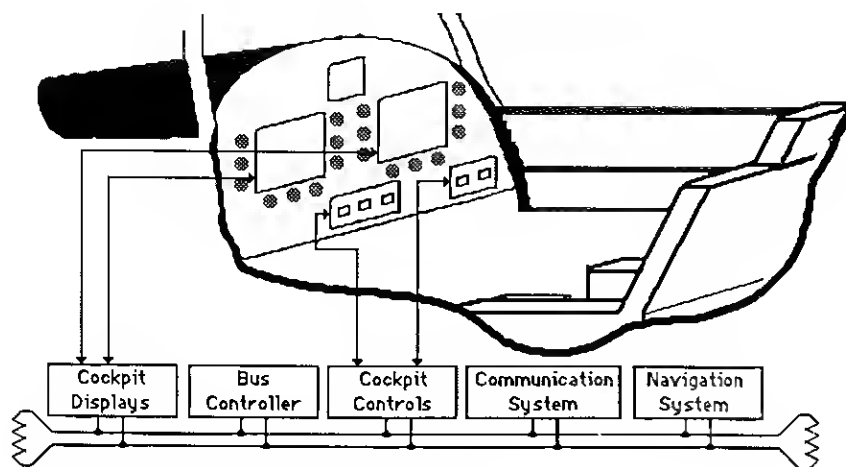


FIGURE 13 Cockpit data bus concept.

presenting flight data to pilots long before the technology was available to support such concepts. Second, a large amount of simulation was performed to test and refine the control and display laws prior to the first flight of these aircraft. (So much simulation now precedes the first flight of a new aircraft that one flight test pilot on return from the first flight of a new fighter is reported to have said, "It flies just like the simulator.")

Another strength that digital technology applies to the crewstation is the use of data buses. The data bus allows all of the different system components to talk to each other and share common data (Fig. 13). This increases the reliability of the entire system by allowing a common component to take over when a similar component fails. For example, if a particular sensor fails, a similar one can take its place and provide data to more than one processor. Likewise, if one display surface fails, another can be used to present the same data to the pilot. Multiple data buses improve reliability by providing additional paths that can be used in case of failure or damage.

Limitations

Most of the limitations in the application of digital technology to the crewstation involve what is still not known. Although the early research provided answers to basic problems of digital crewstations, the actual design process has found many more questions unanswered. These first applications have found that more of the old military human factors standards for crewstation design do not address the digital technology directly. Therefore, a significant amount of both basic and applied research will be needed to make this technology reach its full potential.

A significant problem in the application of electronic displays, especially in a fighter type of cockpit (i.e., an aircraft with a bubble canopy), is the extremely high levels of ambient luminance in the cockpits. The design standard for fighter cockpits requires design for an environment of 10,000 millilamberts. That is approximately equivalent to standing in a field of fresh

snow at noon on a clear day. Such an environment creates a significant problem for those technologies that are not capable of generating the amount of energy that can overcome such an environment (e.g., LED or plasma).

A real potential strength of the digital crewstation may reside in the area of combining data from a number of different sensors, for example, radar, maps or navigation, into one data display. Although there has been some success with small aspects of this problem (e.g., the HSI and ADF discussed earlier and the velocity vector concept described below), there is a significant limitation in the ability to help the pilot combine data from the more broad-band sensors (e.g., radar, infrared). The traditional dedicated display approach basically allows data from different sensors to be presented either serially on the same display surface or in parallel on a number of different display surfaces. The pilot must then use his/her own internal information processing capabilities to mix these inputs into a usable whole. Although people can perform this function, it takes valuable time from the processing of other important flight control processing needs.

Using traditional thinking, there are basically two approaches being applied to this problem (Fig. 14). The first uses a large number of small display surfaces. Although this allows for parallel presentation of a large amount of data to the pilot, little combining can be done on any one surface without serious loss of resolution. The pilot must therefore still do a significant amount of the processing. The second approach puts fewer but larger displays in a cockpit. Because the displays are larger, more data can be combined on a single surface without a significant loss of resolution. But fewer types of data can be presented simultaneously to the crew because there is less room for parallel displays. In the end, the pilot will still mentally combine some of the data from the different displays. A significant amount of research must be done in the area of determining the best ways to combine data digitally (e.g., using an expert system like software) and the best way of presenting the combined data to the pilot.

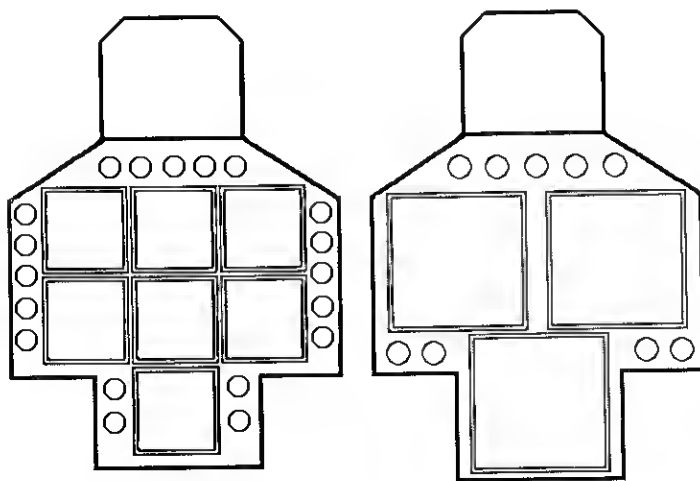


FIGURE 14 Two approaches to the display of large amounts of data.

Hazards

There are two significant hazards in the design of digitally based crewstations. The first involves the very natural temptation to apply this new technology for its own sake or to have an up-to-date high-technology design, rather than selecting the technology that best meets the design needs. The second hazard is almost identical to the first and, yet, the antithesis of it. It involves the use of a digitally based device to essentially mimic the old display technology. This can be seen in some of the CRT-based attitude indicators that simply use the CRT to present an exact copy of the old electromechanical display. Although it may increase reliability, such an approach merely applies the limitations and weaknesses of the older technology-limited formats to a technology that does not have these limitations and is capable of much better display formats.

RECOMMENDED DESIGN APPROACH

The first step in the design of a crewstation using digital technology is the determination of the exact needs of the user. This is accomplished by first defining the types of missions that the system will be designed to meet and then determining which functions will be performed by the human, by the computer, or by a combination of the two. Once the human's functions have been determined, a detailed analysis of the tasks can identify the data and controls needed to perform those tasks. Not only must this analysis create a list of the data needed but it must also determine the relative importance of each of them and interrelationships among them.

There are a number of tools that have proved their effectiveness in making such an analysis. They include analytical methods (e.g., task analysis), simulation studies, and talk-through analyses with experienced pilots. Once determined, the data list is then applied as the driving force in determining the type of display or control technology that should be used.

One example of a successful application of this approach is the velocity vector. Information needs analysis determined that a pilot's real information need in certain types of flight maneuvers, such as instrument flight, was the direct display of the aircraft's actual flight path through the air. When a pilot uses the present-day attitude indicator, he is presented with only a picture of the aircraft's attitude, not the aircraft's flight path. For example, when an aircraft's attitude is 5 degrees nose high, the aircraft may be climbing, flying level, or descending. The actual flight path must be derived by the mental analysis of a number of other instruments. The velocity vector approach does away with this need for unnecessary human processing. When the velocity symbol indicates 5 degrees high, the aircraft is actually climbing at an angle of 5 degrees. When the velocity vector shows the symbol on the horizon, the aircraft is straight and level. The pilot no longer needs to process the input continually from a number of displays to make that determination; the velocity vector does it for him/her.

Test and Retest

Once a concept has been developed, it needs to be tested both statically and dynamically. The static tests involve a task-by-task paper and pencil analysis to make sure that the concept does indeed meet all of the needs

that it was designed to meet. The dynamic tests then ensure that the concept will be effective in a dynamic environment where events and tasks occur in parallel.

The first dynamic method that can be applied is part-task simulation. This involves testing the concept using a simulator that simulates only part of the operational environment. For example, an instrument flight simulator might be used to test the effectiveness of the concept in weather, but such a system could not test its use in a high visibility environment. The part-task simulator is typically used first, because of its relatively low cost.

The next step is to use a simulator that can support the testing of the concept in all of the types of environments and missions that it is designed to support. Here the concept is tested for effectiveness under operational conditions throughout the mission envelope. When problems are identified, they are fixed and retested until they meet their stated objective.

THE FUTURE

The digital crewstation revolution is only in its infancy. Its complete potential for improving the aircraft crewstation is far from known, and only time will tell as to its success. However, several very interesting concepts are being developed. One involves turning the entire instrument panel into one big display surface which, in turn, can have any number of data windows opened at any one time, depending on the current needs of the mission (Fig. 15). This display will, in a very real sense, become a virtual extension of the outside world. It will display a constructed image of the world that lies beyond the nose of the aircraft. Additional data about hazards, navigation paths,

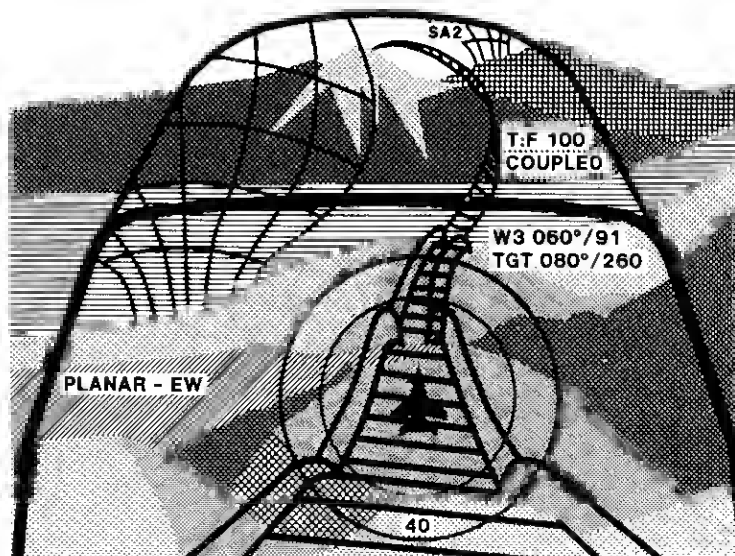


FIGURE 15 Display surface in cockpit.

or other pertinent flight information not normally found in the real world will then be included. Such a system will make flying in zero-zero weather no different from and, in many respects, superior to flying unaided on a clear, sunny day.

Another approach that is being discussed involves the use of the HMD discussed earlier. The HMD would present a virtual display that would include an out-the-window picture overlaid with windows of system data as required or requested by the pilot. The pilot's view would vary as a function of the direction of his/her head. For example, if the pilot looked left, the out-the-window view to the left of the aircraft would be visible and, perhaps, data about the left engine. If the pilot looked right, the out-the-window view to the right of the aircraft would be visible and, perhaps, data about the right engine. The system could be designed so that the pilot could virtually look right through himself/herself and the aircraft and see a virtual image of the external world. The display surface would effectively be a virtual sphere enclosing the pilot.

The digital crewstation has opened up an entirely new design environment in aerospace. It has provided the capability of presenting the pilot with all the information and controls needed to perform the desired mission effectively and safely. The degree to which the technology will, in fact, achieve that goal depends on how well it is applied by the designers and engineers of the future systems.

BIBLIOGRAPHY

- Adam, E. C., et al., *Display Techniques for Advanced Crew Stations*, Air Force Wright Aeronautical Laboratories, AFWAL-TR-84-1016 (AD A146922), 1984.
- AGARD, *Advanced Avionics and the Military Aircraft Man/Machine Interface*, Advisory Group for Aerospace Research and Development, AGARD-CP-329 (AD A119 559), 1982.
- AGARD, *Advancements on Visualization Techniques*, Advisory Group for Aerospace Research and Development, AGARD-CP-255 (AD A094 661), 1980.
- AGARD, *The Guidance and Control of V/STOL Aircraft and Helicopters at Night and in Poor Visibility*, Advisory Group for Aerospace Research and Development, AGARD-CP-148 (AD A013 268), 1974.
- AGARD, *High Speed, Low Level Flight: Aircrew Factors*, Advisory Group for Aerospace Research and Development, AGARD-CP-267 (AD A087 592), 1979.
- ACARD, *Human Factors Consideration in High Performance Aircraft*, Advisory Group for Aerospace Research Development, AGARD-CP-371, 1984.
- AGARD, *Modern Display Technologies and Applications*, Advisory Group for Aerospace Research and Development, AGARD-AR-169, 1982.
- Birt, J. A., and H. T. Task, *A Symposium on Visually Coupled Systems: Development and Application*, Aerospace Medical Division, Brooks AFB, TX, AMD TR-73-1, November 8-10, 1972.
- Chorey, R. A., "Seventy Years of Flight Instruments and Displays," *Aeronaut. J.*, 323-342 (August 1976).
- Ford, T. E., "Flight Deck Display, in *Aircraft Engineering*, March 1983, pp. 2-5.

- Hunt, G. H., "Airborne Electronic Displays, *IEE Proc.*, 128(7), 465-484 (December).
- Jauer, R. A., et al., *Pictorial Formats* (3 volumes), Air Force Wright Aeronautical Laboratories, AFWAL-TR-81-3156, 1982.
- Shager, J. J., *Head-Up Displays: A Literature Review and Analysis with Annotated Bibliography*, U.S. Dept. of Transportation, FAA-RD-78-31, 1978.
- Snyder, H. L., *Human Visual Performance and Flat Panel Display Image Quality*, U.S. Navy Office of Naval Research, ONR-80-1, 1980.
- U.S. Air Force, *Instrument Flying*, U.S. Air Force, Air Training Command, Manual 51-37, 1966.
- U.S. Navy, *Proceedings of the Fifth Advanced Aircrew Display Symposium*, The Naval Air Test Center, Patuxent River, MD, October 1982.
- U.S. Navy, *Proceedings of the Sixth Advanced Aircrew Display Symposium*, The Naval Air Test Center, Patuxent River, MD, May 1984.

JOHN A. WISE

ALGOL

HISTORY

The name ALGOL was derived as an acronym for the International Algorithmic Language but has become a generic term for a whole class of "ALGOL-like" languages. Historically, the language ALGOL 60 was known for a number of years as ALGOL, although it was preceded by ALGOL 58 and various other languages.

In particular, members of the Gesellschaft für Mathematik (GAMM, Society for Applied Mathematics) and Association of Computing Machinery started work on an attempt to define a programming language for mathematical applications which, it was hoped, would supersede existing programming languages. The first version, known as ALGOL 58, was published by Perlis and Samelson at the end of 1958 [1].

Even at this early stage, an attempt was made to formalize the language definition by publishing a detailed reference language, together with a user reference description and some hardware representation details. During the ensuing experimental period, various dialects of ALGOL 58, notably MAD, JOVIAL, NELIAC, ALPHA, and BALGOL, appeared, usually as attempts to remove shortcomings or to add new ideas to the original language. Undoubtedly, the most significant contribution to compiler design and implementation came with an attempt by Backus to give a formal definition of ALGOL 58 [2]. This used a notation that has become known as Backus-Naur or Backus-normal form, normally abbreviated to BNF.

Prior to this, languages were effectively defined by the action of their compilers rather than by a formal definition, which often meant that the only way to verify that a particular construct was acceptable and did what the user expected was to "try it and see." This approach to programming naturally led to disastrous code, which worked under some conditions and not others. It also encouraged compiler writers to build on existing compilers rather than to design complete ones and also to utilize special features of the particular hardware on which they were implementing new or modified languages. This often made new languages difficult or inefficient to port to new machines.

At that time, the formal language definition applied only to the syntax, with usually only informal semantics serving to clarify or merely qualify the syntactic constructs. This had the unfortunate consequence of encouraging some slackness in the syntactic definition with the corresponding deficiencies and ambiguities being clarified by the associated semantics. For example, in one language, the syntax allowed a GOTO statement to lead into a FOR statement, although the semantics, quite rightly, subsequently invalidated this. Despite such problems, the important fact was that a serious attempt had been made to give a formal language definition, and this has led to extensive work in this area.

The ambiguities and redundancies that became apparent in the lifetime of ALGOL 58 led to a new version, ALGOL 60, which was published by Naur (editor) in the "Report on the Algorithmic Language ALGOL 60." Even this was not free from error, and a revised report on the algorithmic language ALGOL 60 was published in 1963. This report defines the language that is now commonly known as ALGOL 60 and, because of its historical significance, is printed as an appendix to this article. It still relies on loosely worded semantics to amplify and clarify the formal syntax.

A precise definition of an ALGOL-like language depends very much on the viewpoint of the user. However, certain base rules can be given. The following were suggested by Morrison in 1979 [3].

1. *Scope rules and block structure.* Names may be introduced to define local quantities and are undefined outside the local environment. However, different environments may use the same name unambiguously to represent different objects.
2. *Abstraction facility.* Abstraction mechanisms may be used to shorten and clarify programs. These are usually in the form of procedures with or without parameters.
3. *Compile time type checking.* The types of all expressions can be checked by a static analysis program.
4. *Infinite store.* The programmer is relieved of the burden of storage allocation and is presented with a conceptually infinite store.
5. *Selective store updating.* This is usually implemented as an efficiency consideration and generally takes the form of an assignment statement.

Clearly, languages such as COBOL, FORTRAN, and BASIC do not meet all these criteria. Languages such as PL/1 and ADA have undoubtedly been strongly influenced by the development of the ALGOL-like languages but involve, in addition, a complexity that is not in sympathy with the simple power of the languages described in the following sections.

ALGOL 60

The basis of ALGOL 60 lies in a block and procedure structure. A block is a clearly defined and delineated ordered sequence of statements associated with the concept of scope of identifiers declared for use within that block. Variables and arrays may be associated with a given block by declaration at the head of the block in which they are to be used and have no existence outside of the block. Scope naturally extends so that nested blocks recognize identifiers declared in outer blocks unless new identifiers of the same name are declared at the head of an inner block. In such a case, the existence of outer identifiers is suspended until the inner block is left, at which time the outer block identifier is revived with its previous value unaltered. That is, the old instance of the identifier comes back into scope. This concept in itself reduces program-checking effort because it is no longer necessary to check whether a name has been used before in a different context. This is particularly useful in validating large programs written by several people.

The efficient implementation of the block concept, which implies a run-time dynamic storage allocation technique, is achieved by a stack mechanism whereby, when a block is entered at run time, a new stack frame containing

the working storage local to the block is created on top of the run-time stack. The current innermost block is thus always on top of the run-time stack and is readily removed by a simple change of stack pointer on leaving the block.

Apart from normal declaration within a block, identifiers may be tagged with the additional declaration "own." This has the effect that on exit from a block, values of "own" identifiers are stored and reinstated on a subsequent entry to the block.

The concept of a procedure is very close to that of a block, except that the procedure header contains the name of the procedure together with an associated set of parameters. The formal parameters in the procedure header are declared along with their own data types in order to allow the associated code of the procedure body to be written independently of any procedure call from the main body of the program. Parameters may either be called by name or by value. In the case of a call by name parameter, the formal parameter is replaced by the corresponding actual parameter from the procedure call. In this case, the value associated with the actual parameter may be changed in the main body of the program as a side effect during the execution of the procedure body. Parameters called by value are handled in a similar way to identifiers declared at the head of a block, except that they are initialized to the value of the corresponding actual parameter, which may be an expression, at call time. Thus, a call by value parameter is equivalent to a new declaration and initialization. Any modification to the value of such parameters is therefore localized to the procedure body and external values are left unchanged.

A procedure may be regarded as a compound statement, along with a set of associated identifiers that are initialized externally. It is thus a generalized statement. Similarly, functions in ALGOL 60 may be regarded as generalized expressions. They also have an associated parameter list but differ from procedures in the sense that they produce a result that is returned as the value of a particular function call.

Data types in ALGOL 60 are clearly defined but few in number. The simple data types are restricted to real, integer, and Boolean. Constant strings are defined but are only used in defining parameter names and are not available to the programmer as a data type.

Arrays of reals, integers, and Booleans are defined and may be of any dimension subject to the limitation of store size in the host computer. The subscript bounds of an array are defined as arithmetic expressions that may, in turn, evaluate to either real or integer values. Clearly, the bounds of the subscripts of an array are essentially of type integer, so real bounds need to be truncated to integers. The flexibility of definition led to compiler inconsistencies and also to programming errors if the programmer did not fully understand the implications of working in real numbers in this context.

In parallel with simple variables, arrays may also be tagged as "own" with corresponding consequences.

An interesting part of the ALGOL 60 language is the control structure. There are various ways in which the order of execution of statements can vary dynamically. The FOR statement of earlier languages is included but is augmented by a WHILE clause, which supplements the terminating value of the FOR statement by an additional Boolean condition. In ALGOL 60, WHILE clauses are not permitted in their own right.

Similarly, the GOTO statement of earlier languages is retained in an extended form. Integer labels are permitted, but symbolic and conditional labels are also permitted. The computed GOTO statement is replaced by a

SWITCH, which associates a list of designated expressions with an identifier. These expressions are implicitly numbered from 1 onward and are selected by subscripting the SWITCH identifier.

Conditional statements are either of the form IF *b* THEN *p* or IF *b* THEN *p* ELSE *q*, where *b* is of type Boolean and *p* and *q* are statements, including blocks, or expressions according to context. The two forms of conditional statement give one- or two-way branching, respectively. Conditional statements may be nested, and this led to an ambiguity in the language that has become known as the "dangling ELSE problem." This problem arises when an IF ... THEN statement follows immediately after the THEN of an IF ... THEN ... ELSE statement. It is then not apparent if the single ELSE relates to the first or second IF. Subsequently, an arbitrary solution to this problem had to be imposed that was esthetically unsatisfactory if practically necessary. This solution, which is equivalent to placing BEGIN, END brackets around the inner IF ... THEN ... ELSE construct associates the final ELSE ... with the innermost corresponding IF.

Despite the shortcomings of the ALGOL 60 definition, it nevertheless was a significant step forward in language design. Many program language designers and modifiers have incorporated ALGOL 60 or subsequent ALGOL-like language constructs in their new languages. Unless the language includes a true block and procedure structure, such modifications are little more than cosmetic and do not achieve the ALGOL aim of simple power.

A side effect of the introduction of ALGOL 60 and subsequent ALGOL-like languages has been the development of a new approach to programming, namely the structured or top-down programming concept. The block, procedure, and control structures of these languages enable programmers to delay detailed thinking until the overall structure of their solution has been determined. This has been one of the most important consequences, the other being the ability to design and implement reliable compilers based on the formal definition of such languages. This is due to the way in which the formal syntax and associated semantics have been utilized directly in the implementation of compilers.

ALGOLW

The originators of ALGOLW were Wirth and Hoare [4], and the first implementation was by a group working under the supervision of Wirth. It took ALGOL 60 as a starting point and attempted to rectify some of the ambiguities and deficiencies of the language.

The simple data types of ALGOL 60 were not only extended to include INT, REAL, and BOOL but also LONGREAL, BIT, and STRING(*n*). STRING does not appear as a data type in its own right but must be modified by a length declaration indicated by an integer literal in parenthesis following the word STRING. This specifies the maximum permitted length of string to be held by the associated variable, with extra characters being packed out with blanks. Logically, this corresponds to an infinity of data types, although in practice these are restricted to a finite number by the size of the host computer. Manipulation of strings may be carried out using the operations of assignment, concatenation, selection of substrings, and comparison.

In addition to the simple data types, there are structured data types, including arrays and records. A record declaration defines a data type consisting of a collection of fields, each with its own declared data type

grouped together under a record name. Objects of a given record class type may be declared by a REFERENCE declaration, which associates a record instance with a given name. Reference expressions may be evaluated and assignments made. Arrays of references are also allowed. List processing can be defined by using references with pointers to references.

Control structures include those found in ALGOL 60, as well as CASE and WHILE statements. The CASE statement is simple, with the control variable being of type INT with the alternative clauses being selected according to the value of the control expression. WHILE statements are controlled by a Boolean expression. The FOR statement includes the possibility of an explicit list of integer values to be taken by the control variable, as well as the usual stepped increment.

Procedures may be either proper or function. A proper procedure compares to a closed procedure, which does not produce an explicit result. Not only are parameters of procedures of a given type but they may also have a qualifier attached. In the simplest form, parameters are called by reference but may also be specified to be called by VALUE, RESULT, or VALUE RESULT. Parameters called by VALUE are evaluated at call time and only this value is passed to the procedure. Call by RESULT parameters similarly pass the final value of the corresponding formal parameter back to the actual parameter on completion of execution of the procedure body. Call by VALUE RESULT is a combination of VALUE and RESULT calls.

Function procedures use the same procedure header as proper procedures, but the procedure body is terminated by an expression that evaluates to the result of the procedure and implicitly determines its type.

Input/output is based on a line or punched card concept. The reserved words READ, READON, WRITE, and WRITEON are the names of input/output functions whose parameters specify the type of information to be expected from the input/output device. The modifier "ON" allows reading to continue on the same line or from the same card as before.

SIMULA 67

The language SIMULA 67 was designed and implemented by Ole-Johan Dahl and others [5]. It contains most of the features of ALGOL 60 as a subset because this basic structure lent itself to extension.

A block may be thought of in two ways. First, it has a static formal existence as a collection of statements structured in a given way. Second, on execution, a dynamic instance of the block is generated. The notion of block instance leads to the possibility of generating several block instances simultaneously. Such block instances may interact in some way. This leads naturally to the concept of a block as a class of objects, each being a dynamic instance of a block with some interactive mechanism. This, in turn, leads to the idea of an object in SIMULA 67 as a self-contained program having its own local data and actions defined by a class declaration. In the case in which no actions are defined, a class of pure data structure results.

New classes may be defined in terms of subclasses, and so on, leading to a powerful structure that is entirely in sympathy with the logical structure of ALGOL 60. In addition to class and subclass definitions, a class concatenation mechanism is introduced so as to be able to define compound objects.

A consequence of the class concept is that users may build data structures and programs to operate on them, thus producing problem-orientated application languages.

As the name SIMULA 67 suggests, the idea of simulation languages was close to the design aims of the languages designers, so one possible class would be "simulation," which would contain subclass declarations corresponding to the various subtasks required in the simulation and programs to link them together.

Because many instances of a class may exist at any time, it is necessary to be able to distinguish between them. SIMULA 67 uses the list concept to do this with both logical simplicity and some of the power of list processing.

New data types CHARACTER and TEST enable string handling to be implemented. Input/output definitions were not given in ALGOL 60 on the grounds that these, being machine dependent, were not generally definable. SIMULA 67, however, takes a different view based on the file as an input/output primitive. Although this does not solve all of the problems of input/output, it nevertheless avoids the incompatibility problems of different implementations of ALGOL 60 arising from the lack of such guidelines.

The "own" declarations of ALGOL 60 are not included in SIMULA 67 because by the time of its implementation, the practical difficulties of "own" declarations were well known. Pieces of information could either be referred to by value, which essentially means itself, or by reference, which corresponds to a pointer to the information. Procedure parameters may also be called by name, as in ALGOL 60.

SIMULA 67 made significant advances from ALGOL 60. It has perhaps paradoxically suffered from its name, which usually associates it with simulation processes, whereas its application area is far wider and more general than this. It has certainly contributed significantly to language theory and design and is in many ways superior to some of the later ALGOL 60 derivatives.

ALGOL 68

ALGOL 68 was designed by Working Group 2.1 of the International Federation for Information Processing. It was formally defined, using two-level grammars, in a report edited by van Wijngaarden et al. [6]. Experience with the use of the language led to revisions, and a revised report was published by van Wijngaarden et al. in 1975 [7]. Works by Lindsey and van der Meulen [8], and Meertens and van Vliet [9] accompanied the two reports to give an informal account of ALGOL 68.

This article refers to the revised rather than the original report. It cannot be a detailed account but discusses some of the aspects of ALGOL 68, which make it very different from earlier ALGOL-like languages. The structural aspects of programs and data are at the heart of its design. The highly recursive nature of the language definition makes difficult reading, and it was for this reason that the books by the authors mentioned above [8, 9] were published simultaneously. For example, the idea of an enclosed clause is fundamental and is defined as either a closed, conditional, case, conformity, or collateral clause. Each of these is defined in turn, often recursively, and this makes it difficult for the novice, or indeed the experienced nonacademic programmer, to come to grips with the language. It is quite possible, however, to learn to use ALGOL 68 from introductory texts [10] without ever needing to

understand the underlying generality, although serious programmers would need to do this if they were to utilize its full power.

A fundamental concept in ALGOL 68 is that of the "mode" of an object, which loosely compares to type in ALGOL 60. Thus, modes INT (integer), REAL, CHAR (character), and BOOL (Boolean) are simple modes, but more complex modes are required for more complex data objects. The user is free to define more complex data objects with associated modes built up from modes already defined either in the basic language or by the user. Mode checking is performed by the system in relation to the associated operands. A direct disparity in mode does not necessarily imply an error. For example, if an object of mode INT is found where one of type REAL is expected, the principle of coercion is used to coerce mode INT to mode REAL. This simple example, which corresponds to normal practice in many other languages, is only one case of the general principle of coercion or widening in ALGOL 68. In some contexts, the strength of coercion is greater than in others. Thus, a strong context accepts only one mode of object which, in turn, causes all possible types of coercion to be applied to the given object in order to try to make it fit the required strong mode.

As in ALGOL 60, objects may be grouped together to form arrays of one or more dimensions. These new objects must therefore have an associated mode that includes the dimensionality of the array. Thus, a three-dimensional array of reals has mode row, row, row of REAL, or [, ,] REAL, for short. Note that the word row here means dimension rather than the usual row of an array.

A new grouping of data objects into a structure is introduced. A structure contains fields of specified modes. Thus,

```
STRUCT (INT this, REAL that)
```

declares a structure mode with an integer field with name *this* and a real field with name *that*. The mode itself may be given a name for future use. Thus,

```
MODE thing = STRUCT (INT this, REAL that)
```

is valid.

In addition to declaration of new modes, ALGOL 68 allows declarations of new, or additions to old, operators. This is done in conjunction with the mode or modes of the operands of the given operator. The declaration of an operator or modification to an existing one uses the reserved word OP and needs to declare the modes of the operands and of the result together with a code description of the action of the operator itself.

ALGOL 68 includes the usual declaration of variables in a block structure but makes a clear distinction between a variable and the value to which it refers (REF). This enables the mode of the variable itself, which is REF INT, to be distinguished from the mode (INT) of the object to which it refers. This is useful when elements of different arrays or structures refer to the same objects, in which case, copies of the object do not need to be made if the elements or fields are declared to be of mode REF. Once the initial shock of thinking in this way has been overcome, it leads to clearer thinking on the part of the programmer. A combination of MODE, STRUCT, and REF allows a list processing system to be built.

The control structures of ALGOL 60 have been widened to include several forms of a WHILE loop and a CASE statement. The WHILE clause includes a Boolean control expression that determines whether or not a specified series of statements is to be executed repeatedly. The CASE statement is a generalized form of the IF . . . THEN . . . ELSE clause, which enables one of a series of alternatives to be selected for execution, depending on the value of some control expression.

In ALGOL 68, the procedure and function declarations of ALGOL 60 are covered by a single technique. Not only is a declaration of names and modes of the parameters included in the procedure header but an additional mode specifying the type of result is also included. In the case of an ALGOL 60 type procedure, the mode of the corresponding ALGOL 68 procedure is VOID. The modes of the parameters can be of the simple variety or of type REF, which enables structures to be passed without copying. This is particularly valuable in the case of large structures. In addition, procedures may be passed as parameters of procedures themselves.

The word "transput" is used synonymously with input/output. Transput commands are on files that, by default, correspond to the standard transput devices for the system. However, a number of transput channels may be available to the user. Three standard channels, namely character, formatted character, and binary, are provided with the system. These are known as transputs without format. Formatted transputs include BIT, Boolean, number, string, and character. These, along with a file declaration, result in a powerful input/output or, rather, transput system.

ALGOL 68 has significantly advanced the understanding of the underlying principles of language design [11-13]. Initial compilers were written with considerable difficulty, and many of the earlier ones were not complete implementations of the full language. Because of its intrinsic complexity, ALGOL 68 has not been used as a basis for language extension as ALGOL 60 has.

PASCAL

The language Pascal was developed by Wirth in the late 1960s, and a formal description was published in 1971 [14]. Wirth declared two principal aims for the language. The first was to provide a language to enable programming to be taught as a systematic discipline and to emphasize clearly and concisely the fundamental concepts on which it was based. The second was to dispel the notion that useful nontrivial languages must be slow in either compilation or execution and that their compilers must always contain errors.

Pascal is another language based on ALGOL 60 although, as usual, ALGOL 60 is not itself a subset of Pascal. It was also considerably influenced by ALGOLW. The main extensions are related to data structuring and to control structures.

There are five scalar data types, namely INT, REAL, BOOL, CHAR, and ALFA. ALFA type values are sequences of characters that are not directly accessible themselves.

A scalar type may also be a subrange of another scalar type, defined by indicating the smallest and largest values of the subrange.

Structured data types fall into five classes, namely, arrays, records, power sets, files, and classes. Arrays are defined as in ALGOL 60 but may also be defined as packed arrays where appropriate. Records are similar to

those defined in ALGOLW. A power set is a set of values that is the power set of its base type. A file may be in one of three modes, namely input, output, and neutral, and modes may be switched in the course of processing a file. At any one time, only one item of a file is accessible. The class definition allows a class of components of the same type, whose number may alter during the execution of a program. In particular, use of class structures enables the construction of data corresponding to finite graphs by use of a pointer type that is associated with each variable of class type.

Control structures include a case statement that is controlled by a selector expression, which selects the one statement whose constant label has the same value as the selector. Repetitive loops may be controlled by either WHILE or REPEAT commands, thus enabling a loop to be safely broken either at the beginning, end, or the middle of the loop. There is also a FOR statement.

Procedures and functions are similar to those in ALGOLW. However, procedure and function parameters are classified into constant, variable, procedure, and function parameters. A constant parameter corresponds to a call by value, whereas a variable parameter corresponds to a call by name. Procedure and function parameters specify the names of procedures or functions to be called.

Not only do operators in Pascal include the usual arithmetic, Boolean, relational, and string handling operators but they also include the set operators union, intersection, difference, and membership for operation on power sets.

Pascal is widely used as a teaching and applications language. It has been criticized both formally by Habbermann [15] and by Lecarme and Desjardins [16] and practically by users whose criticisms are usually of particular implementations that, for example, severely restrict the use of power sets, and also for the incompatibilities between implementations that inhibit portability of Pascal programs. The ludicrous situation is that despite the formal description of Pascal as given by Wirth [14], a standard for the language was introduced more than 10 years later to define a common base for most of the implementations.

S-ALGOL

S-ALGOL is a programming language designed by Morrison [3] and informally described in a book by Cole and Morrison [13]. It has been implemented on a number of microcomputers and also on some mainframe machines. It has not been widely recognized as a programming language, although it is in use as a teaching language in a few universities and schools. It is described briefly here because it is the basis from which the more general language PS-ALGOL was implemented. It was designed as a teaching language but is also used as a simple, general-purpose programming language.

Three ground rules were used as a basis for the language design, namely

1. *The principle of correspondence* [17–19]. Roughly this principle requires that the rules governing names be designed together. This applies to scope as well as to declarations and parametric construction. For example, in Pascal, types may be declared, but not passed, as parameters.

2. *The principle of abstraction* [17-19]. This principle involves the process of extracting the general structure of objects as an abstraction. For example, the mathematical notion of function is an abstraction over expression.
3. *The principle of data type completeness* [18]. Effectively, all data types must have the same "civil rights," and rules for using data types should be complete. This rule is violated in ALGOLW, where arrays are not allowed as fields of records, and in Pascal, where only some objects are allowed as elements of sets.

In S-ALGOL, the application of these rules has led to a clean language that helps the programmer to concentrate on solving the problem rather than on twisting the problem to match the vagaries of some programming language. One deficiency, which is rectified in PS-ALGOL, is the lack of procedure-producing procedures, which contradicts the principle of data type completeness.

Since S-ALGOL is not widely used, its syntax and semantics will not be described in detail here. However, some general consequences of the application of the rules given above may be of some interest. Once a concept has been introduced, it must be applicable anywhere that it makes sense. Thus, for example, once arrays have been declared, arrays of arrays must be allowed and assignment of arrays and slices of arrays permitted. Similarly, instances of structures must be allowed as elements of arrays and must also be allowed to be passed as parameters of procedures. Primitives, such as the graphics picture (pic), which involves "point" as a simple data object, are data types that must be capable of declaration, assignment, and passing as parameters or results of procedure calls.

Literals and variables are distinguished clearly and may not change their status. Strings are strings, rather than arrays of characters, with the consequence that there is one rather than a multiplicity of data types string (n).

Input/output is based on the file concept. Objects may be placed in or taken from a file. The type of object expected is implicit in the appropriate read statement.

To ensure the principle of data type completeness, the totality of data objects is given by the closure of the scalar data types under the recursive application of one-dimensional array, or vector, and structure constructs.

The brief description given above may imply an academic abstractness that is not present in S-ALGOL as a practical programming language. Its control structures are simple but powerful; its string handling is elegant. It is certainly worthy of study by ALGOL-like language designers and by teachers of programming methodology.

PS-ALGOL

PS-ALGOL is in the process of development as a language to provide the facilities normally left to other functions such as the file system or loader. The language designers [3, 20] aim to provide a totally integrated environment so that the user never has to abandon the language for any computational activity. The first PS-ALGOL compiler was written in 1981 using S-ALGOL as an implementation tool. In this account, the detail of the language will not be discussed but only the general principles that are invoked in the attempt to meet the aim mentioned above.

The name PS-ALGOL is an abbreviation for persistent S-ALGOL. Persistence is used in relation to the length of time that a data object exists. In traditional languages, data do not last longer than the period of activation of the program without the explicit use of some storage agency, such as a file store or data base system. The objective of persistent programming is to allow data to outlive the program with uniform access methods whether the lifetime of the data is short or long term.

As with S-ALGOL, PS-ALGOL ensures data type completeness by the closure of data types under the recursive application of vector and structure constructions. Procedures must also be included in the process of closure if they are to be first-class citizens in the sense of Strachey [18]. The closure of a procedure is all the information required to execute the procedure correctly, namely the code together with the part of its environment needed to execute it. The principle of abstraction must also be applied to define the abstract data type of a procedure. The machinery for abstraction of data objects and procedures must be the same.

The persistent store of PS-ALGOL consists entirely of legal PS-ALGOL data objects, which include procedures. The store is partitioned into data bases to allow concurrency control and protection. The data base is organized on root data structure principles, which allows access to other data objects in the same data base. Because procedures are first-class data types that can return procedures as results, it follows that abstract data objects must be implemented. Also, modules or ADA packages may be simulated.

Pictures are included as data types, either by type picture for line drawings or by type image for bit maps. The persistent store is available for pictures and images and, in addition, by having data types for graphics, syntactic support can be given for the manipulation of these objects by infix operators.

The PS-ALGOL system provides a persistent store facility, together with a secure transaction mechanism. The persistence concept in itself does not provide rich environmental facilities; this is dependent on the richness of the underlying programming language. One of the aims of the PS-ALGOL group is to provide this richness in conjunction with completeness.

CONCLUSION

The ALGOL-like programming languages have made a substantial contribution to the understanding of the use of programming languages both by sophisticated and novice users. In particular, the ways of defining reference language descriptions that originated with the BNF descriptions of ALGOL 58 and ALGOL 60 have developed and become an important tool in language design. The goal of clean language design is now much nearer than it was in 1960, with a consequent easing of the sheer intellectual effort required to write large or complex programs. The whole area of top-down or structured programming is a direct result of the logical structure underlying ALGOL language design. To a lesser extent, this approach has been copied in an attempt to make programming easier in other traditional and new languages. The subject is by no means closed.

REFERENCES

1. A. J. Perlis and K. Samelson, "Preliminary Report: International Algebraic Language," *Comm. ACM*, 1(12) 8-22 (December 1958).
2. J. W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," *Proc. Int. Conf. Inf. Process.*, Unesco, Paris, 1959.
3. R. M. Morrison, "On the Development of Algol," Ph.D thesis, University of St. Andrews, Scotland, 1979.
4. N. Wirth and C. A. R. Hoare, "A Contribution to the Development of ALGOL," *Comm. ACM*, 9(6) 413-431 (1966).
5. Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard, *Simula 67: Common Base Language*, Norwegian Computing Center, Oslo, 1968.
6. A. van Wijngaarden et al., "Report on the Algorithmic Language ALGOL 68," *Numer. Math.* 14, 79 (1969).
7. A. van Wijngaarden et al., "Revised Report on the Algorithmic Language ALGOL 68," *Acta Inf.* 5 1-234 (1975).
8. C. H. Lindsey and S. G. van der Meulen, *Informal Introduction to ALGOL 68*, North Holland, Amsterdam, 1971.
9. L. G. Meertens and J. C. van Vliet, *A Syntax-Directed Error Recovery Method for Parsing ALGOL 68 Programs: Report in 6/73*, Mathematical Center, Amsterdam, 1973.
10. D. F. Brailsford and A. N. Walker, *Introductory ALGOL 68 Programming*, Ellis Horwood, Chichester, United Kingdom; John Wiley, New York, 1979.
11. "Information Processing 93," in *Proceedings of the IFIP Ninth World Computer Congress*, Paris, 1983.
12. J. C. Van Vliet and H. Wupper, eds., *Proceedings of the International Conference on ALGOL 68*, Amsterdam, 1981.
13. A. J. Cole and R. Morrison, *An Introduction to Programming with S-ALGOL*, Cambridge University Press, Cambridge, United Kingdom, 1983.
14. N. Wirth, "The Programming Language Pascal," *Acta Ing.* 1 35-63 (1971).
15. A. N. Habbermann, "Critical Comments on the Programming Language Pascal," *Acta Inf.* 3 47-57 (1973).
16. O. Lecarme and P. Desjardins, "More Comments on the Programming Language Pascal," *Acta Inf.* 4 231-243 (1975).
17. P. J. Landin, "A Correspondence between ALGOL 60 and Church's Lambda Notation," *Comm. ACM*, 8(2-3) 89-101, 158-165 (1965).
18. C. Strachey, *Fundamental Concepts in Programming Languages*, Oxford University Press, Oxford, United Kingdom, 1967.
19. R. D. Tennant, "Language Design Methods Based on Semantic Principles," *Acta Inf.*, 8, 97-112 (1977).
20. M. P. Atkinson et al., "An Approach to Persistent Programming," *Comp. J.*, 26(4), 360-365 (November 1983).

A. J. COLE

APPENDIX A

Revised Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS
F. L. BAUER
J. GREEN

C. KATZ
J. MCCARTHY
A. J. PERLS

H. RUTISHAUSER
K. SAMELSON
B. VAUQUOIS

J. H. WEGSTEIN
A. VAN WIJNGAARDEN
M. WOODGER

Dedicated to the Memory of WILLIAM TURANSKI

SUMMARY

The report gives a complete defining description of the international algorithmic language ALGOL 60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition, the notions, reference language, publication language and hardware representations are explained.

In the first chapter, a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic units known as identifiers, numbers and strings are defined. Further, some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical) and designational.

The fourth chapter describes the operational units of the language, known as statements. The basic statements are: assignment statements (evaluation of a formula), go to statements (explicit break of the sequence of execution of statements), dummy statements, and procedure statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: conditional statements, for statements, compound statements, and blocks.

In the fifth chapter, the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with two detailed examples of the use of the language and an alphabetic index of definitions.

CONTENTS

INTRODUCTION

1. STRUCTURE OF THE LANGUAGE
 - 1.1. Formalism for syntactic description
2. BASIC SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS.
 - BASIC CONCEPTS.
 - 2.1. Letters
 - 2.2. Digits. Logical values.
 - 2.3. Delimiters
 - 2.4. Identifiers
 - 2.5. Numbers
 - 2.6. Strings
 - 2.7. Quantities, kinds and scopes
 - 2.8. Values and types
3. EXPRESSIONS
 - 3.1. Variables
 - 3.2. Function designators
 - 3.3. Arithmetic expressions
 - 3.4. Boolean expressions
 - 3.5. Designational expressions
4. STATEMENTS
 - 4.1. Compound statements and blocks
 - 4.2. Assignment statements
 - 4.3. Go to statements
 - 4.4. Dummy statements
 - 4.5. Conditional statements
 - 4.6. For statements
 - 4.7. Procedure statements
5. DECLARATIONS
 - 5.1. Type declarations
 - 5.2. Array declarations
 - 5.3. Switch declarations
 - 5.4. Procedure declarations

EXAMPLES OF PROCEDURE DECLARATIONS

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

INTRODUCTION

Background

After the publication of a preliminary report on the algorithmic language ALGOL,^{1,2} as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an *ALGOL Bulletin*, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959 which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represent the following organizations: Association Française de Calcul, British Computer Society, Gesellschaft für Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the *Communications of the ACM*, where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the *ACM Communications*. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

January 1960 Conference

The thirteen representatives,³ from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960.

Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur

and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

April 1962 Conference [Edited by M. Woodger]

A meeting of some of the authors of ALGOL 60 was held on April 2-3, 1962 in Rome, Italy, through the facilities and courtesy of the International Computation Centre. The following were present:

Authors	Advisers	Observer
F. L. Bauer	M. Paul	W. L. van der Poel
J. Green	R. Franciotti	(Chairman, IFIP)
C. Katz	P. Z. Ingerman	TC 2.1 Working
R. Kogon		Group ALGOL)
(representing J. W. Backus)		
P. Naur		
K. Samelson	G. Seegmüller	
J. H. Wegstein	R. E. Utman	
A. van Wijngaarden		
M. Woodger	P. Landin	

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the ALGOL 60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification that were submitted by interested parties in response to the Questionnaire in *ALGOL Bulletin* No. 14 were used as a guide.

This report* constitutes a supplement to the ALGOL 60 Report which should resolve a number of difficulties therein. Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced pro-

* [Editor's Note. The present edition follows the text which was approved by the Council of IFIP. Although it is not clear from the Introduction, the present version is the original report of the January 1960 conference modified according to the agreements reached during the April 1962 conference. Thus the report mentioned here is incorporated in the present version. The modifications touch the original report in the following sections: Changes of text: 1 with footnote; 2.1 footnote; 2.3; 2.7; 3.3.3; 3.3.4.2; 4.1.3; 4.2.3; 4.2.4; 4.3.4; 4.7.3; 4.7.3.1; 4.7.3.3; 4.7.5.1; 4.7.5.4; 4.7.6; 5; 5.3.3; 5.3.5; 5.4.3; 5.4.4; 5.4.5. Changes of syntax: 3.4.1; 4.1.1; 4.2.1; 4.5.1.]

¹ Preliminary report—International Algebraic Language. *Comm. ACM* 1, 12 (1958), 8.

² Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, edited by A. J. Perlis and K. Samelson. *Num. Math.* 1 (1959), 41-60.

³ William Turanski of the American group was killed by an automobile just prior to the January 1960 Conference.

gramming languages will lead to better resolution:

1. Side effects of functions
2. The call by name concept
3. **own**: static or dynamic
4. For statement: static or dynamic
5. Conflict between specification and declaration

The authors of the ALGOL 60 Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962 and has been approved by the Council of the International Federation for Information Processing.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

REFERENCE LANGUAGE

1. It is the working language of the committee.
2. It is the defining language.
3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.⁵
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.

DESCRIPTION OF THE REFERENCE LANGUAGE

Was sich überhaupt sagen lässt, lässt
sich klar sagen; und wovon man nicht
reden kann, darüber muss man schweigen.
LEONARD WITTMANNSTEIN.

1. Structure of the Language

As stated in the introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication—and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition,

7. The main publications of the ALGOL language itself will use the reference representation.

PUBLICATION LANGUAGE

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g., subscripts, spaces, exponents, Greek letters).
2. It is used for stating and communicating processes.
3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

HARDWARE REPRESENTATIONS

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

For transliteration between the reference language and a language suitable for publications, among others, the following rules are recommended.

Reference Language	Publication Language
Subscript bracket []	Lowering of the line between the brackets and removal of the brackets
Exponentiation ↑	Raising of the exponent
Parentheses ()	Any form of parentheses, brackets, braces
Basis of ten 10	Raising of the ten and of the following integral number, inserting of the intended multiplication sign

self-contained units of the language—explicit formulae—called assignment statements.

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe, e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refer to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets **begin** and **end** to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an

array of numbers, or even the set of rules defining a function. A sequence of declarations followed by a sequence of statements and enclosed between **begin** and **end** constitutes a block. Every declaration appears in a block in this way and is valid only for that block.

A program is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it.

In the sequel the syntax and semantics of the language will be given.⁴

1.1. FORMALISM FOR SYNTACTIC DESCRIPTION

The syntax will be described with the aid of metalinguistic formulae.⁵ Their interpretation is best explained by an example

$(ab) ::= (| | (ab) (| (ab) (d))$

Sequences of characters enclosed in the brackets $\langle \rangle$ represent metalinguistic variables whose values are sequences of symbols. The marks $::=$ and $|$ (the latter with the meaning of **or**) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable (ab) . It indicates that (ab) may have the value $($ or $|$ or that given some legitimate value of (ab) , another may be formed by following it with the character $($ or by following it with some value of the variable (d) . If the values of (d) are the decimal digits, some values of (ab) are:

$(((((1(37($
 $(12345($
 $(($
 $(86$

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets $\langle \rangle$) as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition:

$(empty) ::=$
(i.e. the null string of symbols).

⁴ Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is left undefined or said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

⁵ Cf. J. W. Backus, The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. Proc. Internat. Conf. Inf. Proc., UNESCO, Paris. June 1959.

2. Basic Symbols, Identifiers, Numbers, and Strings. Basic Concepts.

The reference language is built up from the following basic symbols:

$\langle \text{basic symbol} \rangle ::= \langle \text{letter} \rangle \langle \text{digit} \rangle \langle \text{logical value} \rangle \langle \text{delimiter} \rangle$

2.1. LETTERS

$\langle \text{letter} \rangle ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|$
 $A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z$

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e. character not coinciding with any digit, logical value or delimiter).

Letters do not have individual meaning. They are used for forming identifiers and strings⁶ (cf. sections 2.4. Identifiers, 2.6. Strings).

2.2.1. DIGITS

$\langle \text{digit} \rangle ::= 0|1|2|3|4|5|6|7|8|9$

Digits are used for forming numbers, identifiers, and strings.

2.2.2. LOGICAL VALUES

$\langle \text{logical value} \rangle ::= \text{true}|\text{false}$

The logical values have a fixed obvious meaning.

2.3. DELIMITERS

$\langle \text{delimiter} \rangle ::= \langle \text{operator} \rangle \langle \text{separator} \rangle \langle \text{bracket} \rangle \langle \text{declarator} \rangle$
 $\langle \text{specifier} \rangle$
 $\langle \text{operator} \rangle ::= \langle \text{arithmetic operator} \rangle \langle \text{relational operator} \rangle$
 $\langle \text{logical operator} \rangle \langle \text{sequential operator} \rangle$
 $\langle \text{arithmetic operator} \rangle ::= +|-|\times|/|++|$
 $\langle \text{relational operator} \rangle ::= <|<=|>|>=|$
 $\langle \text{logical operator} \rangle ::= \equiv|\supseteq|\vee|\wedge|\neg$
 $\langle \text{sequential operator} \rangle ::= \text{go to}|\text{if}|\text{then}|\text{else}|\text{for}|\text{do}^7$
 $\langle \text{separator} \rangle ::= ,|;|:|:=|\text{step}|\text{until}|\text{while}|\text{comment}$
 $\langle \text{bracket} \rangle ::= ()|[]|{}|\text{'begin'end}$
 $\langle \text{declarator} \rangle ::= \text{own}|\text{Boolean}|\text{integer}|\text{real}|\text{array}|\text{switch}|\text{procedure}$
 $\langle \text{specifier} \rangle ::= \text{string}|\text{label}|\text{value}$

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

For the purpose of including text among the symbols of

⁶ It should be particularly noted that throughout the reference language underlining [in typewritten copy; boldface type in printed copy—Ed.] is used for defining independent basic symbols (see sections 2.2.2 and 2.3). These are understood to have no relation to the individual letters of which they are composed. Within the present report [not including headings—Ed.], boldface will be used for no other purpose.

⁷ **do** is used in for statements. It has no relation whatsoever to the **do** of the preliminary report, which is not included in ALGOL 60.

a program the following "comment" conventions hold:

The sequence of basic symbols: *is equivalent to*

```

; comment (any sequence not containing ;) ;
begin comment (any sequence not containing ;) ; begin
end (any sequence not containing end or ; or else) end

```

By equivalence is here meant that any of the three structures shown in the left-hand column may be replaced, in any occurrence outside of strings, by the symbol shown on the same line in the right-hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.4. IDENTIFIERS

2.4.1. Syntax

(identifier) ::= (letter)|(identifier)(letter)|(identifier)(digit)

2.4.2. Examples

```

  q
Soup
V17a
a34kTMNs
MARILYN

```

2.4.3. Semantics

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf., however, section 3.2.4. Standard Functions).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7. Quantities, Kinds and Scopes, and section 5. Declarations).

2.5. NUMBERS

2.5.1. Syntax

```

(unsigned integer) ::= (digit)|(unsigned integer)(digit)
(integer) ::= (unsigned integer)|(+(unsigned integer)|
              -(unsigned integer))
(decimal fraction) ::= .(unsigned integer)
(exponent part) ::= 10(integer)
(decimal number) ::= (unsigned integer)|(decimal fraction)|
                    (unsigned integer)(decimal fraction)
(unsigned number) ::= (decimal number)|(exponent part)|
                    (decimal number)(exponent part)
(number) ::= (unsigned number)|(+(unsigned number)|
              -(unsigned number))

```

2.5.2. Examples

0	-200.084	-.083 ₁₀ -02
177	+07.43 ₁₀ 8	- ₁₀ 7
.5384	9.34 ₁₀ +10	₁₀ -4
+0.7300	2- ₁₀ 4	+ ₁₀ +5

2.5.3. Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4. Types

Integers are of type **integer**. All other numbers are of type **real** (cf. section 5.1. Type Declarations).

2.6. STRINGS

2.6.1. Syntax

```

(proper string) ::= (any sequence of basic symbols not containing
                    ' or ')|(empty)
(open string) ::= (proper string)|(open string)'
                (open string)(open string)
(string) ::= ' (open string) '

```

2.6.2. Examples

```

'5k.,- '[[ 'Λ = /: 'Tt'
'.. This u is u a u 'string'

```

2.6.3. Semantics

In order to enable the language to handle arbitrary sequences of basic symbols the string quotes ' and ' are introduced. The symbol *u* denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. sections 3.2. Function Designators and 4.7. Procedure Statements).

2.7. QUANTITIES, KINDS AND SCOPES

The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures.

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

2.8. VALUES AND TYPES

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

The various "types" (**integer**, **real**, **Boolean**) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

```

(expression) ::= (arithmetic expression)|(Boolean expression)|
                (designational expression)

```

3.1. VARIABLES

3.1.1. Syntax

```

(variable identifier) ::= (identifier)
(simple variable) ::= (variable identifier)
(subscript expression) ::= (arithmetic expression)
(subscript list) ::= (subscript expression) [(subscript list),
    (subscript expression)]
(array identifier) ::= (identifier)
(subscripted variable) ::= (array identifier) [(subscript list)]
(variable) ::= (simple variable) [(subscripted variable)]

```

3.1.2. Examples

```

epsilon
delA
a17
Q[7,2]
z[sin(n×pi/2),Q[3,n,4]]

```

3.1.3. Semantics

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1. Type Declarations) or for the corresponding array identifier (cf. section 5.2. Array Declarations).

3.1.4. Subscripts

3.1.4.1. Subscripted variables designate values which are components of multidimensional arrays (cf. section 5.2. Array Declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets []: The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3. Arithmetic Expressions).

3.1.4.2. Each subscript position acts like a variable of type integer and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2. Array Declarations).

3.2. FUNCTION DESIGNATORS

3.2.1. Syntax

```

(procedure identifier) ::= (identifier)
(actual parameter) ::= (string) [(expression) [(array identifier)]
    (switch identifier) [(procedure identifier)]
(letter string) ::= (letter) [(letter string) (letter)]
(parameter delimiter) ::= , [(letter string) : {
(actual parameter list) ::= (actual parameter) {
    (actual parameter list) (parameter delimiter)
    (actual parameter)
(actual parameter part) ::= (empty) [(actual parameter list)]
(function designator) ::= (procedure identifier)
    (actual parameter part)

```

3.2.2. Examples

```

sin(a-b)
J(v+s,n)
R
S(s-5)Temperature:(T)Pressure:(P)
Compile(' := ')Stack:(Q)

```

3.2.3. Semantics

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4. Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7. Procedure Statements. Not every procedure declaration defines the value of a function designator.

3.2.4. Standard functions

Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. It is recommended that this reserved list should contain:

<i>abs</i> (E)	for the modulus (absolute value) of the value of the expression E
<i>sign</i> (E)	for the sign of the value of E (+1 for E>0, 0 for E=0, -1 for E<0)
<i>sqr</i> (E)	for the square root of the value of E
<i>sin</i> (E)	for the sine of the value of E
<i>cos</i> (E)	for the cosine of the value of E
<i>arctan</i> (E)	for the principal value of the arctangent of the value of E
<i>ln</i> (E)	for the natural logarithm of the value of E
<i>exp</i> (E)	for the exponential function of the value of E (e ^E).

These functions are all understood to operate indifferently on arguments both of type real and integer. They will all yield values of type real, except for *sign*(E) which will have values of type integer. In a particular representation these functions may be available without explicit declarations (cf. section 5. Declarations).

3.2.5. Transfer functions

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely,

entier(E),

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E.

3.3. ARITHMETIC EXPRESSIONS

3.3.1. Syntax

```

(adding operator) ::= +|-
(multiplying operator) ::= ×|/|+
(primary) ::= (unsigned number) [(variable)]
    (function designator) [(arithmetic expression)]
(factor) ::= (primary) [(factor) ↑ (primary)]
(term) ::= (factor) [(term) (multiplying operator) (factor)]
(simple arithmetic expression) ::= (term) {
    (adding operator) (term) [(simple arithmetic expression)]
    (adding operator) (term)
(if clause) ::= if (Boolean expression) then
    (arithmetic expression) ::= (simple arithmetic expression) |
    (if clause) (simple arithmetic expression) else
    (arithmetic expression)

```

3.3.2. Examples

Primaries:

```
7.39410 - 8
sum
w[i+2,8]
cos(y+z×3)
(a-3/y+vu↑8)
```

Factors:

```
omega
sum↑cos(y+z×3)
7.39410 - 8↑w[i+2,8]↑(a-3/y+vu↑8)
```

Terms:

```
U
omega×sum↑cos(y+z×3)/7.39410 - 8↑w[i+2,8]↑
(a-3/y+vu↑8)
```

Simple arithmetic expression:

```
U - Yu + omega×sum↑cos(y+z×3)/7.39410 - 8↑w[i+2,8]↑
(a-3/y+vu↑8)
```

Arithmetic expressions:

```
w×u - Q(S+Cu)↑2
if q>0 then S+3×Q/A else 2×S+3×q
if a<0 then U+V else if a×b>17 then U/V else if
  k≠y then V/U else 0
a×sin(omega×t)
0.5712×a[N×(N-1)/2, 0]
(A×arctan(y)+Z)↑(7+Q)
if q then n-1 else n
if a<0 then A/B else if b=0 then B/A else z
```

3.3.3. Semantics

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4.4. Values of Function Designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4. Boolean Expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value **true** is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position

is understood). The construction:

else (simple arithmetic expression)

is equivalent to the construction:

else if true then (simple arithmetic expression)

3.3.4. Operators and types

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types **real** or **integer** (cf. section 5.1. Type Declarations). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1. The operators $+$, $-$, and \times have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be **integer** if both of the operands are of **integer** type, otherwise **real**.

3.3.4.2. The operations (term)/(factor) and (term) ÷ (factor) both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

$$a/b \times 7 / (p-q) \times v/s$$

means

$$(((a \times (b^{-1})) \times 7) \times ((p-q)^{-1}) \times v) \times (s^{-1})$$

The operator $/$ is defined for all four combinations of types **real** and **integer** and will yield results of **real** type in any case. The operator \div is defined only for two operands both of type **integer** and will yield a result of type **integer**, mathematically defined as follows:

$$a \div b = \text{sign } (a/b) \times \text{entier}(\text{abs}(a/b))$$

(cf. sections 3.2.4 and 3.2.5).

3.3.4.3. The operation (factor)↑(primary) denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example,

$$2 \uparrow n \uparrow k \quad \text{means} \quad (2^n)^k$$

while

$$2 \uparrow (n \uparrow m) \quad \text{means} \quad 2^{(n^m)}$$

Writing i for a number of **integer** type, r for a number of **real** type, and a for a number of either **integer** or **real** type, the result is given by the following rules:

- $a \uparrow i$ If $i > 0$, $a \times a \times \dots \times a$ (i times), of the same type as a .
 If $i = 0$, if $a \neq 0, 1$, of the same type as a .
 if $a = 0$, undefined.
 If $i < 0$, if $a \neq 0$, $1/(a \times a \times \dots \times a)$ (the denominator has $-i$ factors), of type **real**.
 if $a = 0$, undefined.
- $a \uparrow r$ If $a > 0$, $\exp(r \times \ln(a))$, of type **real**.
 If $a = 0$, if $r > 0, 0.0$, of type **real**.
 if $r \leq 0$, undefined.
 If $a < 0$, always undefined.

3.3.5. Precedence of operators

The sequence of operations within one expression is

generally from left to right, with the following additional rules:

3.3.5.1. According to the syntax given in section 3.3.1 the following rules of precedence hold:

first: \uparrow
second: \times/\div
third: $+-$

3.3.5.2. The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6. Arithmetics of real quantities

Numbers and variables of type **real** must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4. BOOLEAN EXPRESSIONS

3.4.1. Syntax

(relational operator) ::= $<|\leq|=|\geq|>|\neq$
(relation) ::= (simple arithmetic expression)
(relational operator)(simple arithmetic expression)
(Boolean primary) ::= (logical value)|(variable)|
(function designator)|(relation)|((Boolean expression))
(Boolean secondary) ::= (Boolean primary)| \neg |(Boolean primary)
(Boolean factor) ::= (Boolean secondary)|
(Boolean factor) \wedge |(Boolean secondary)|
(Boolean term) ::= (Boolean factor)|(Boolean term)
 \vee |(Boolean factor)|
(implication) ::= (Boolean term)|(implication) \supset |(Boolean term)
(simple Boolean) ::= (implication)|
(simple Boolean) = (implication)
(Boolean expression) ::= (simple Boolean)|
(if clause)(simple Boolean) **else** (Boolean expression)

3.4.2. Examples

```
x = -2
Y > Z ∨ z < q
a + b > -5 ∧ z - d > q / 2
p ∧ q ∨ x ≠ y
g = ¬a ∧ b ∧ ¬c ∨ d ∨ e ⊃ ¬f
if k < 1 then s > w else h ≤ c
if if a then b else c then d else f then g else h < k
```

3.4.3. Semantics

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4. Types

Variables and function designators entered as Boolean

primaries must be declared **Boolean** (cf. section 5.1. Type Declarations and section 5.4.4. Values of Function Designators).

3.4.5. The operators

Relations take on the value **true** whenever the corresponding relation is satisfied for the expressions involved, otherwise **false**.

The meaning of the logical operators \neg (not), \wedge (and), \vee (or), \supset (implies), and $=$ (equivalent), is given by the following function table.

b1	false	false	true	true
b2	false	true	false	true
$\neg b1$	true	true	false	false
$b1 \wedge b2$	false	false	false	true
$b1 \vee b2$	false	true	true	true
$b1 \supset b2$	true	true	false	true
$b1 = b2$	true	false	false	true

3.4.6. Precedence of operators

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1. According to the syntax given in section 3.4.1 the following rules of precedence hold:

first: arithmetic expressions according to section 3.3.5.
second: $< \leq = \geq > \neq$
third: \neg
fourth: \wedge
fifth: \vee
sixth: \supset
seventh: $=$

3.4.6.2. The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.5. DESIGNATIONAL EXPRESSIONS

3.5.1. Syntax

(label) ::= (identifier)|(unsigned integer)
(switch identifier) ::= (identifier)
(switch designator) ::= (switch identifier)|(subscript expression)|
(simple designational expression) ::= (label)|(switch designator)|
((designational expression))
(designational expression) ::= (simple designational expression)|
(if clause)(simple designational expression) **else** (designational expression)

3.5.2. Examples

```
17
p9
Choose[n-1]
Town[if y < 0 then N else N+1]
if Ab < c then 17 else q[if w ≤ 0 then 2 else n]
```

3.5.3. Semantics

A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. section 5.3.

Switch Declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4. The subscript expression

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1, 2, 3, ..., n, where n is the number of entries in the switch list.

3.5.5. Unsigned integers as labels

Unsigned integers used as labels have the property that leading zeros do not affect their meaning, e.g. 00217 denotes the same label as 217.

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1. COMPOUND STATEMENTS AND BLOCKS

4.1.1. Syntax

```
(unlabelled basic statement) ::= (assignment statement) |
  (go to statement) | (dummy statement) | (procedure statement)
(basic statement) ::= (unlabelled basic statement) | (label):
  (basic statement)
(unconditional statement) ::= (basic statement) |
  (compound statement) | (block)
(statement) ::= (unconditional statement) |
  (conditional statement) | (for statement)
(compound tail) ::= (statement) end | (statement) ;
  (compound tail)
(block head) ::= begin (declaration) (block head) ;
  (declaration)
(unlabelled compound) ::= begin (compound tail)
(unlabelled block) ::= (block head) ; (compound tail)
(compound statement) ::= (unlabelled compound) |
  (label):(compound statement)
(block) ::= (unlabelled block) | (label):(block)
(program) ::= (block) | (compound statement)
```

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

```
L: L: ... begin S ; S ; ... S ; S end
```

Block:

```
L: L: ... begin D ; D ; .. D ; S ; S ; ... S ;
  S end
```

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2. Examples

Basic statements:

```
a := p+q
go to Naples
START: CONTINUE: W := 7.993
```

Compound statement:

```
begin x := 0 ; for y := 1 step 1 until n do
  x := x+A[y] ;
  if x>q then go to STOP else if x>w-2 then
    go to S ;
Aw: S: W := x+bob end
```

Block:

```
Q: begin integer i, k ; real w ;
  for i := 1 step 1 until m do
    for k := i+1 step 1 until m do
      begin w := A[i, k] ;
        A[i, k] := A[k, i] ;
        A[k, i] := w end for i and k
  end block Q
```

4.1.3. Semantics

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. section 5. Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be nonlocal to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e. the smallest block whose brackets begin and end enclose that statement. In this context a procedure body must be considered as if it were enclosed by begin and end and treated as a block.

Since a statement of a block may again itself be a block the concepts local and nonlocal to a block must be understood recursively. Thus an identifier, which is nonlocal to a block A, may or may not be nonlocal to the block B in which A is one statement.

4.2. ASSIGNMENT STATEMENTS

4.2.1. Syntax

```
(left part) ::= (variable) := | (procedure identifier) :=
(left part list) ::= (left part) | (left part list) (left part)
(assignment statement) ::= (left part list) (arithmetic expression) |
  (left part list) (Boolean expression)
```

4.2.2. Examples

```

s := p[0] := n := n+1+s
n := n+1
A := B/C-v-qXS
S[v,k+2] := 3-arcTan(sXzeta)
V := Q>Y^Z

```

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4. Types

The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is **Boolean**, the expression must likewise be **Boolean**. If the type is **real** or **integer**, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from **real** to **integer** type, the transfer function is understood to yield a result equivalent to

$$\text{entier}(E+0.5)$$

where E is the value of the expression. The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4).

4.3. Go To STATEMENTS

4.3.1. Syntax

(go to statement) ::= go to (designational expression)

4.3.2. Examples

```

go to 8
go to exit (n+1)
go to Town [if y < 0 then N else N+1]
go to if Ab < c then 17 else q [if w < 0 then 2 else n]

```

4.3.3. Semantics

A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4. Restriction

Since labels are inherently local, no go to statement can lead from outside into a block. A go to statement may, however, lead from outside into a compound statement.

4.3.5. Go to an undefined switch designator

A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4. DUMMY STATEMENTS

4.4.1. Syntax

(dummy statement) ::= (empty)

4.4.2. Examples

```

L:
begin ... ; John; end

```

4.4.3. Semantics

A dummy statement executes no operation. It may serve to place a label.

4.5. CONDITIONAL STATEMENTS

4.5.1. Syntax

```

(if clause) ::= if (Boolean expression) then
(unconditional statement) ::= (basic statement);
(compound statement)|(block)
(if statement) ::= (if clause) (unconditional statement)
(conditional statement) ::= (if statement)|(if statement) else
(statement)|(if clause)(for statement)|
(label) : (conditional statement)

```

4.5.2. Examples

```

if x > 0 then n := n+1
if v > u then V: q := n+m else go to R
if s < 0 ∨ P ≤ Q then AA: begin if q < v then a := v/s
                        else y := 2×a end
                        else if v > s then a := v-q else if v > s-1
                        then go to S

```

4.5.3. Semantics

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1. If statement. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

```
if B1 then S1 else if B2 then S2 else S3 ; S4
```

and

```
if B1 then S1 else if B2 then S2 else if B3 then S3 ; S4
```

Here $B1$ to $B3$ are Boolean expressions, while $S1$ to $S3$ are unconditional statements. $S4$ is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expression of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value **true** is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be $S4$, i.e. the state-

ment following the complete conditional statement. Thus the effect of the delimiter **else** may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

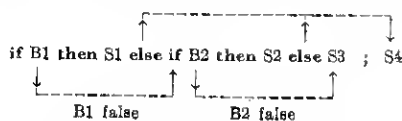
```
else (unconditional statement)
```

is equivalent to

```
else if true then (unconditional statement)
```

If none of the Boolean expressions of the **if** clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:



4.5.4. Go into a conditional statement

The effect of a **go to** statement leading into a conditional statement follows directly from the above explanation of the effect of **else**.

4.6. FOR STATEMENTS

4.6.1. Syntax

```

(for list element) ::= (arithmetic expression) |
  (arithmetic expression) step (arithmetic expression) until
  (arithmetic expression) | (arithmetic expression) while
  (Boolean expression)
(for list) ::= (for list element) [(for list), (for list element)]
(for clause) ::= for (variable) := (for list) do
(for statement) ::= (for clause) (statement) |
  (label) : (for statement)
  
```

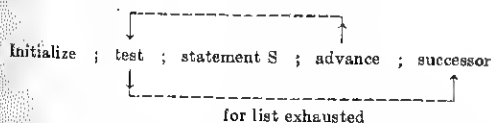
4.6.2. Examples

```

for q := 1 step 2 until n do A[q] := B[q]
for k := 1, V1×2 while V1 < N do
  for j := I+G, L, 1 step 1 until N, C+D do
    A[k,j] := B[k,j]
  
```

4.6.3. Semantics

A **for** clause causes the statement **S** which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



In this picture the word **initialize** means: perform the first assignment of the **for** clause. **Advance** means: perform the next assignment of the **for** clause. **Test** determines if the last assignment has been done. If so, the execution con-

tinues with the successor of the **for** statement. If not, the statement following the **for** clause is executed.

4.6.4. The for list elements

The **for** list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the **for** list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of **for** list elements and the corresponding execution of the statement **S** are given by the following rules:

4.6.4.1. Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement **S**.

4.6.4.2. Step-until-element. An element of the form **A step B until C**, where **A**, **B**, and **C**, are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

```

V := A ;
L1: if (V-C) × sign(B) > 0 then go to element exhausted;
    statement S ;
    V := V+B ;
    go to L1 ;
  
```

where **V** is the controlled variable of the **for** clause and **element exhausted** points to the evaluation according to the next element in the **for** list, or if the **step-until-element** is the last of the list, to the next statement in the program.

4.6.4.3. While-element. The execution governed by a **for** list element of the form **E while F**, where **E** is an arithmetic and **F** a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

```

L3: V := E ;
    if ¬F then go to element exhausted ;
    Statement S ;
    go to L3 ;
  
```

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit

Upon exit out of the statement **S** (supposed to be compound) through a **go to** statement the value of the controlled variable will be the same as it was immediately preceding the execution of the **go to** statement.

If the exit is due to exhaustion of the **for** list, on the other hand, the value of the controlled variable is undefined after the exit.

4.6.6. Go to leading into a for statement

The effect of a **go to** statement, outside a **for** statement, which refers to a label within the **for** statement, is undefined.

4.7. PROCEDURE STATEMENTS

4.7.1. Syntax

```

(actual parameter) ::= (string) | (expression) | (array identifier) |
  (switch identifier) | (procedure identifier)
(letter string) ::= (letter) | (letter string) (letter)
  
```



```

(parameter delimiter) ::= , | (letter string) ;
(actual parameter list) ::= (actual parameter) |
    (actual parameter list) (parameter delimiter)
    (actual parameter)
(actual parameter part) ::= (empty) |
    ((actual parameter list))
(procedure statement) ::= (procedure identifier)
    (actual parameter part)

```

4.7.2. Examples

```

Spur (A) Order: (7) Result to: (V)
Transpose (W, v+1)
Absmax (A, N, M, Yg, I, K)
Innerproduct (A [i, P, u], B [P], 10, P, Y)

```

These examples correspond to examples given in section 5.4.2.

4.7.3. Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. Procedure Declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. Value assignment (call by value)

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and Types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.2. Name replacement (call by name)

Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3. Body replacement and execution

Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any nonlocal quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.7.4. Actual-formal correspondence

The correspondence between the actual parameters of

the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.

This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4. A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5. Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6. Deleted.

4.7.7. Parameter delimiters

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the

same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed in code

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the **begin**, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through **end**, or by a **go to** statement) all identifiers which are declared for the block lose their local significance.

A declaration may be marked with the additional declarator **own**. This has the following effect: upon a re-entry into the block, the values of **own** quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as **own** are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax.

```
(declaration) ::= (type declaration)|(array declaration)|
                 (switch declaration)|(procedure declaration)
```

5.1. TYPE DECLARATIONS

5.1.1. Syntax

```
(type list) ::= (simple variable)|(
                 (simple variable), (type list)
  (type) ::= real | integer | Boolean
  (local or own type) ::= (type)|own (type)
  (type declaration) ::= (local or own type)(type list)
```

5.1.2. Examples

```
integer p,q,s
own Boolean Acryl,n
```

5.1.3. Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values

including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values **true** and **false**.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of **own**, see the fourth paragraph of section 5 above.

5.2. ARRAY DECLARATIONS

5.2.1. Syntax

```
(lower bound) ::= (arithmetic expression)
(upper bound) ::= (arithmetic expression)
(bound pair) ::= (lower bound):(upper bound)
(bound pair list) ::= (bound pair)|(bound pair list),(bound pair)
(array segment) ::= (array identifier)|(bound pair list)|
                    (array identifier),(array segment)
(array list) ::= (array segment)|(array list),(array segment)
(array declaration) ::= array (array list)|(local or own type)
                    array (array list)
```

5.2.2. Examples

```
array a, b, c[7:n,2:m], s[-2:10]
own integer array A[if c<0 then 2 else 1:20]
real array q[-7:-1]
```

5.2.3. Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

5.2.3.1. Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter **:**. The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3. Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type **real** is understood.

5.2.4. Lower upper bound expressions

5.2.4.1 The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2).

5.2.4.2. The expressions can only depend on variables and procedures which are nonlocal to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4. The expressions will be evaluated once at each entrance into the block.

5.2.5. The identity of subscripted variables

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. How-

ever, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3. SWITCH DECLARATIONS

5.3.1. Syntax

```
(switch list) ::= {designational expression}!
(switch list), (designational expression)
(switch declaration) ::= switch (switch identifier) := (switch list)
```

5.3.2. Examples

```
switch S := S1, S2, Q[u], if v > -5 then S3 else S4
switch Q := p1, w
```

5.3.3. Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, ..., obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5. Designational Expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4. Evaluation of expressions in the switch list

An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5. Influence of scopes

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4. PROCEDURE DECLARATIONS

5.4.1. Syntax

```
(formal parameter) ::= (identifier)
(formal parameter list) ::= (formal parameter)
(formal parameter list) (parameter delimiter)
(formal parameter)
(formal parameter part) ::= (empty) | ((formal parameter list))
(identifier list) ::= (identifier) (identifier list), (identifier)
(value part) ::= value (identifier list) ; | (empty)
(specifier) ::= string[(type)] array[(type) array] label switch
procedure[(type) procedure
(specification part) ::= (empty) | (specifier) (identifier list) ; |
(specification part) (specifier) (identifier list) ;
(procedure heading) ::= (procedure identifier)
(formal parameter part) ; (value part) (specification part)
(procedure body) ::= (statement) (code)
(procedure declaration) ::=
procedure (procedure heading) (procedure body) |
(type) procedure (procedure heading) (procedure body)
```

5.4.2. Examples (see also the examples at the end of the report)

```
procedure Spur(a) Order:(n) Result:(s) ; value n ;
array a ; integer n ; real s ;
begin integer k ;
s := 0 ;
for k := 1 step 1 until n do s := s + a[k,k]
end
```

```
procedure Transpose(a) Order:(n) ; value n ;
array a ; integer n ;
begin real w ; integer i, k ;
for i := 1 step 1 until n do
for k := 1 + i step 1 until n do
begin w := a[i,k] ;
a[i,k] := a[k,i] ;
a[k,i] := w
end
end Transpose
```

```
integer procedure Step(u) ; real u ;
Step := if 0 ≤ u ∧ u ≤ 1 then 1 else 0
```

```
procedure Absmax(a) size:(n,m) Result:(y) Subscripts:(i,k) ;
comment The absolute greatest element of the matrix a,
of size n by m is transferred to y, and the subscripts of this
element to i and k ;
array a ; integer n, m, i, k ; real y ;
begin integer p, q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until m do
if abs(a[p,q]) > y then begin y := abs(a[p,q]) ; i := p ;
k := q
end end Absmax
```

```
procedure Innerproduct(a,b) Order:(k,p) Result:(y) ; value k ;
integer k,p ; real y,a,b ;
begin real s ;
s := 0 ;
for p := 1 step 1 until k do s := s + a × b ;
y := s
end Innerproduct
```

5.4.3. Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. section 3.2. Function Designators and section 4.7. Procedure Statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or nonlocal to the body depending on whether they are declared within the body or not. Those of them which are nonlocal to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a

block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.4. Values of function designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.5. Specifications

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.6. Code as procedure body

It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language

Examples of Procedure Declarations:

EXAMPLE 1.

```

procedure euler (fct, sum, eps, tim) ; value eps, tim ;
integer tim ; real procedure fct ; real sum, eps ;
comment euler computes the sum of fct(i) for i from zero up to
infinity by means of a suitably refined euler transformation. The
summation is stopped as soon as tim times in succession the absolute
value of the terms of the transformed series are found to be
less than eps. Hence, one should provide a function fct with one
integer argument, an upper bound eps, and an integer tim. The
output is the sum sum. euler is particularly efficient in the case
of a slowly convergent or divergent alternating series ;
begin integer i, k, n, t ; array m[0:15] ; real mn, mp, ds ;
i := n := t := 0 ; m[0] := fct(0) ; sum := m[0]/2 ;
nextterm: i := i+1 ; mn := fct(i) ;
for k := 0 step 1 until n do
  begin mp := (mn+m[k])/2 ; m[k] := mn ;
  mn := mp end means ;

```

```

if (abs(mn) < abs(m[n])) and (n < 15) then
  begin ds := mn/2 ; n := n+1 ; m[n] :=
    mn end accept
else ds := mn ;
sum := sum + ds ;
if abs(ds) < eps then t := t+1 else t := 0 ;
if t < tim then go to nextterm

```

end euler

EXAMPLE 2.*

```

procedure RK(x,y,n,FKT,eps,eta,xE,yE,f) ; value x,y ;
integer n ; Boolean f ; real x,eps,eta,xE ; array
y,yE ; procedure FKT ;
comment RK integrates the system  $y'_k = f_k(x, y_1, y_2, \dots, y_n)$ 
( $k=1, 2, \dots, n$ ) of differential equations with the method of Runge-
Kutta with automatic search for appropriate length of integration
step. Parameters are: The initial values x and y[k] for x and the un-
known functions yk(x). The order n of the system. The procedure
FKT(x,y,n,x) which represents the system to be integrated, i.e.
the set of functions fk. The tolerance values eps and eta which
govern the accuracy of the numerical integration. The end of the
integration interval xE. The output parameter yE which repre-
sents the solution at x=xE. The Boolean variable f, which must
always be given the value true for an isolated or first entry into
RK. If however the functions y must be available at several mesh-
points x0, x1, ..., xn, then the procedure must be called repeatedly
(with x=x0, xE=xk+1, for k=0, 1, ..., n-1) and then the
later calls may occur with f=false which saves computing time.
The input parameters of FKT must be x,y,n, the output parameter
z represents the set of derivatives z[k]=fk(x,y[1],y[2], ..., y[n])
for x and the actual y's. A procedure comp enters as a nonlocal
identifier ;

```

begin

```

array z,y[1],y[2],y[3][1:n] ; real z1,x2,x3,H ; Boolean out ;
integer k,j ; own real s,Hs ;
procedure RK1ST(x,y,h,xE,ye) ; real x,h,xE ; array
y,ye ;

```

comment RK1ST integrates one single RUNGE-KUTTA
with initial values z,y[k] which yields the output
parameters xe=x+h and ye[k], the latter being the
solution at xe. Important: the parameters n, FKT, z
enter RK1ST as nonlocal entities ;

begin

```

array w[1:n],a[1:5] ; integer k,j ;
a[1] := a[2] := a[5] := h/2 ; a[3] := a[4] := h ;
xe := x ;
for k := 1 step 1 until n do ye[k] := w[k] := y[k] ;
for j := 1 step 1 until 4 do

```

begin

```

  FKT(xe,w,n,z) ;
  ze := x+a[j] ;
  for k := 1 step 1 until n do
    begin
      w[k] := y[k]+a[j]*z[k] ;
      ye[k] := ye[k] + a[j+1]*z[k]/3
    

```

* This RK-program contains some new ideas which are related to ideas of S. GILL, A process for the step-by-step integration of differential equations in an automatic computing machine, [Proc. Camb. Phil. Soc. 47 (1951), 96]; and E. FRÖBERG, On the solution of ordinary differential equations with digital computing machines, [Fysiograf. Sällsk. Lund, Förh. 20, 11 (1950), 136-152]. It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

```

    end k
  end ;
end RK1ST ;
Begin of program:
  if f then begin H := xE - x ; s := 0 end else H := Hs ;
  out := false ;
AA: if (x + 2.01 × H - xE > 0) == (H > 0) then
  begin Hs := H ; out := true ; H := (xE - x) / 2
  end if ;
  RK1ST (x, y, 2 × H, x1, y1) ;
BB: RK1ST (x, y, H, x2, y2) ; RK1ST (x2, y2, H, x3, y3) ;
  for k := 1 step 1 until n do
    if comp(y1[k], y3[k], eta) > eps then go to CC ;

```

```

comment: comp(a, b, c) is a function designator, the value
of which is the absolute value of the difference of the
mantissae of a and b, after the exponents of these quan-
tities have been made equal to the largest of the exponents
of the originally given parameters a, b, c :
x := x3 ; if out then go to DD ;
for k := 1 step 1 until n do y[k] := y3[k] ;
if s = 5 then begin s := 0 ; H := 2 × H end if ;
s := s + 1 ; go to AA ;
CC: H := 0.5 × H ; out := false ; x1 := x2 ;
for k := 1 step 1 until n do y1[k] := y2[k] ;
go to BB ;
DD: for k := 1 step 1 until n do yE[k] := y3[k]
end RK

```

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

All references are given through section numbers. The references are given in three groups:

def Following the abbreviation "def", reference to the syntactic definition (if any) is given.

synt Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words (in typewritten copy; boldface in printed copy—Ed.) have been collected at the beginning.

The examples have been ignored in compiling the index.

+, see: plus
 -, see: minus
 ×, see: multiply
 /, +, see: divide
 †, see: exponentiation
 <, ≤, =, ≥, >, ≠, see: (relational operator)
 =, ⊃, ∨, ∧, ¬, see: (logical operator)
 ,, see: comma
 ., see: decimal point
 n, see: ten
 :, see: colon
 ;, see: semicolon
 :=, see: colon equal
 u, see: space
 (), see: parentheses
 [], see: subscript brackets
 ', see: string quotes

(actual parameter), def 3.2.1, 4.7.1
 (actual parameter list), def 3.2.1, 4.7.1
 (actual parameter part), def 3.2.1, 4.7.1
 (adding operator), def 3.3.1
 alphabet, text 2.1
 arithmetic, text 3.3.6
 (arithmetic expression), def 3.3.1 synt 3, 3.1.1, 3.3.1, 3.4.1, 4.2.1, 4.6.1, 5.2.1 text 3.3.3
 (arithmetic operator), def 2.3 text 3.3.4
 array, synt 2.3, 5.2.1, 5.4.1
 array, text 3.1.4.1
 (array declaration), def 5.2.1 synt 5 text 5.2.3
 (array identifier), def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
 (array list), def 5.2.1
 (array segment), def 5.2.1
 (assignment statement), def 4.2.1 synt 4.1.1 text 1, 4.2.3

(basic statement), def 4.1.1 synt 4.5.1
 (basic symbol), def 2
 begin, synt 2.3, 4.1.1
 (block), def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
 (block head), def 4.1.1
 Boolean, synt 2.3, 5.1.1 text 5.1.3

(Boolean expression), def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1 text 3.4.3
 (Boolean factor), def 3.4.1
 (Boolean primary), def 3.4.1
 (Boolean secondary), def 3.4.1
 (Boolean term), def 3.4.1
 (bound pair), def 5.2.1
 (bound pair list), def 5.2.1
 (bracket), def 2.3

(code), synt 5.4.1 text 4.7.8, 5.4.6
 colon :, synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
 colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1
 comma ,, synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1
 comment, synt 2.3
 comment convention, text 2.3
 (compound statement), def 4.1.1 synt 4.5.1 text 1
 (compound tail), def 4.1.1
 (conditional statement), def 4.5.1 synt 4.1.1 text 4.5.3

(decimal fraction), def 2.5.1
 (decimal number), def 2.5.1 text 2.5.3
 decimal point ., synt 2.3, 2.5.1
 (declaration), def 5 synt 4.1.1 text 1, 5 (complete section)
 (declarator), def 2.3
 (delimiter), def 2.3 synt 2
 (designational expression), def 3.5.1 synt 3, 4.3.1, 5.3.1 text 3.5.3
 (digit), def 2.2.1 synt 2, 2.4.1, 2.5.1
 dimension, text 5.2.3.2
 divide / +, synt 2.3, 3.3.1 text 3.3.4.2
 do, synt 2.3, 4.6.1
 (dummy statement), def 4.4.1 synt 4.1.1 text 4.4.3

else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.5.1 text 4.5.3.2
 (empty), def 1.1 synt 2.6.1, 3.2.1, 4.4.1, 4.7.1, 5.4.1
 end, synt 2.3, 4.1.1
 entier, text 3.2.5
 exponentiation †, synt 2.3, 3.3.1 text 3.3.4.3.
 (exponent part), def 2.5.1 text 2.5.3
 (expression), def 3 synt 3.2.1, 4.7.1 text 3 (complete section)

- (factor), def 3.3.1
- false, synt 2.2.2
- for, synt 2.3, 4.6.1
- (for clause), def 4.6.1 text 4.6.3
- (for list), def 4.6.1 text 4.6.4
- (for list element), def 4.6.1 text 4.6.4.1, 4.6.4.2, 4.6.4.3
- (formal parameter), def 5.4.1 text 5.4.3
- (formal parameter list), def 5.4.1
- (formal parameter part), def 5.4.1
- (for statement), def 4.6.1 synt 4.1.1, 4.5.1 text 4.6 (complete section)
- (function designator), def 3.2.1 synt 3.3.1, 3.4.1 text 3.2.3, 5.4.4
- go to, synt 2.3, 4.3.1
- (go to statement), def 4.3.1 synt 4.1.1 text 4.3.3
- (identifier), def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1 text 2.4.3
- (identifier list), def 5.4.1
- if, synt 2.3, 3.3.1, 4.5.1
- (if clause), def 3.3.1, 4.5.1 synt 3.4.1, 3.5.1 text 3.3.3, 4.5.3.2
- (if statement), def 4.5.1 text 4.5.3.1
- (implication), def 3.4.1
- integer, synt 2.3, 5.1.1 text 5.1.3
- (integer), def 2.5.1 text 2.5.4
- label, synt 2.3, 5.4.1
- (label), def 3.5.1 synt 4.1.1, 4.5.1, 4.6.1 text 1, 4.1.3
- (left part), def 4.2.1
- (left part list), def 4.2.1
- (letter), def 2.1 synt 2, 2.4.1, 3.2.1, 4.7.1
- (letter string), def 3.2.1, 4.7.1
- local, text 4.1.3
- (local or own type), def 5.1.1 synt 5.2.1
- (logical operator), def 2.3 synt 3.4.1 text 3.4.5
- (logical value), def 2.2.2 synt 2, 3.4.1
- (lower bound), def 5.2.1 text 5.2.4
- minus —, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
- multiply X, synt 2.3, 3.3.1 text 3.3.4.1
- (multiplying operator), def 3.3.1
- nonlocal, text 4.1.3
- (number), def 2.5.1 text 2.5.3, 2.5.4
- (open string), def 2.6.1
- (operator), def 2.3
- own, synt 2.3, 5.1.1 text 5, 5.2.5
- (parameter delimiter), def 3.2.1, 4.7.1 synt 5.4.1 text 4.7.7
- (parentheses ()), synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 4.7.1, 5.4.1 text 3.3.5.2
- plus +, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
- (primary), def 3.3.1
- procedure, synt 2.3, 5.4.1
- (procedure body), def 5.4.1
- (procedure declaration), def 5.4.1 synt 5 text 5.4.3
- (procedure heading), def 5.4.1 text 5.4.3
- (procedure identifier) def 3.2.1 synt 3.2.1, 4.7.1, 5.4.1 text 4.7.5.
- (procedure statement), def 4.7.1 synt 4.1.1 text 4.7.3
- (program), def 4.1.1 text 1
- (proper string), def 2.6.1
- quantity, text 2.7
- real, synt 2.3, 5.1.1 text 5.1.3
- (relation), def 3.4.1 text 3.4.5
- (relational operator), def 2.3, 3.4.1
- scope, text 2.7
- semicolon ;, synt 2.3, 4.1.1, 5.4.1
- (separator), def 2.3
- (sequential operator), def 2.3
- (simple arithmetic expression), def 3.3.1 text 3.3.3
- (simple Boolean), def 3.4.1
- (simple designational expression), def 3.5.1
- (simple variable), def 3.1.1 synt 5.1.1 text 2.4.3
- space u, synt 2.3 text 2.3, 2.6.3
- (specification part), def 5.4.1 text 5.4.5
- (specifier), def 2.3
- (specifier), def 5.4.1
- standard function, text 3.2.4, 3.2.5
- (statement), def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)
- statement bracket, see: begin end
- step, synt 2.3, 4.6.1 text 4.6.4.2
- string, synt 2.3, 5.4.1
- (string), def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
- string quotes ' ', synt 2.3, 2.6.1, text 2.6.3
- subscript, text 3.1.4.1
- subscript bound, text 5.2.3.1
- subscript brackets [], synt 2.3, 3.1.1, 3.5.1, 5.2.1
- (subscripted variable), def 3.1.1 text 3.1.4.1
- (subscript expression), def 3.1.1 synt 3.5.1
- (subscript list), def 3.1.1
- successor, text 4
- switch, synt 2.3, 5.3.1, 5.4.1
- (switch declaration), def 5.3.1 synt 5 text 5.3.3
- (switch designator), def 3.5.1 text 3.5.3
- (switch identifier), def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
- (switch list), def 5.3.1
- (term), def 3.3.1
- ten 10, synt 2.3, 2.5.1
- then, synt 2.3, 3.3.1, 4.5.1
- transfer function, text 3.2.5
- true, synt 2.2.2
- (type), def 5.1.1 synt 5.4.1 text 2.8
- (type declaration), def 5.1.1 synt 5 text 5.1.3
- (type list), def 5.1.1
- (unconditional statement), def 4.1.1, 4.5.1
- (unlabelled basic statement), def 4.1.1
- (unlabelled block), def 4.1.1

(unlabelled compound), def 4.1.1
 (unsigned integer), def 2.5.1, 3.5.1
 (unsigned number), def 2.5.1 synt 3.3.1
 until, synt 2.3, 4.6.1 text 4.6.4.2
 (upper bound), def 5.2.1 text 5.2.4
 value, synt 2.3, 5.4.1

value, text 2.8, 3.3.3
 (value part), def 5.4.1 text 4.7.3.1
 (variable), def 3.1.1 synt 3.3.1, 3.4.1, 4.2.1, 4.6.1 text 3.1.3
 (variable identifier), def 3.1.1
 while, synt 2.3, 4.6.1 text 4.6.4.3

END OF THE REPORT

NOTE: This Report is published in the *Communications of the ACM*, in *Numerische Mathematik*, and in *The Computer Journal*. Reproduction of this Report for any purpose is explicitly permitted; reference should be made to this issue of *Communications* and to the respective issues of *Numerische Mathematik* and *The Computer Journal* as the source.
 Reprints are available as follows from the Association for Computing Machinery, 211 East 43 Street, New York 17, N.Y.:
 Single copies to individuals, no charge; Single copies to companies, 50 cts.; Multiple copies: first ten, 50 cts. ea.; next 25 cts. ea.; all over 100, 10 cts. ea.

ALPHA MICROSYSTEMS

COMPANY PROFILE

Alpha Microsystems designs, manufactures, markets, and services multiuser computer systems and software for business, professional, scientific, engineering, and educational markets.

Alpha Micro was founded in 1977 by Robert B. Hitchcock, currently President and Chairman of the Board and Richard C. Wilcox, now Vice President and Chief Technical Officer.

The company's world headquarters are located in Santa Ana, California.

COMPANY OVERVIEW

Alpha Micro History—1977 to Present

Founded and incorporated in 1977, Alpha Micro was the first company to offer a multiuser, multitasking microcomputer system, a major technological breakthrough resulting in computers with superior price/performance.

Alpha Micro has continued to be a leader with innovative technological developments, including file-oriented video cassette recorder backup and simultaneous data broadcasting via satellite to remote microcomputers. Alpha Micro was also one of the first companies to produce a family of systems based on the Motorola 68000 microprocessor and to offer a broad range of Winchester storage capacities.

Today the company's broad and widely varied product line includes a complete family of MC68000, MC68010, and MC68020 multiuser systems; an add-on board, which converts an IBM PC/XT and compatibles into a multiuser Alpha Micro system; a full line of peripherals; an innovative video-based product line; communication options; high-level languages and development tools; and applications software.

Products are distributed through a network of over 470 value added resellers (VARs), dealers, subdealers, and systems houses serving thousands of Alpha Micro users worldwide. Wholly owned subsidiaries in Great Britain, Belgium, and Australia provide sales, service, and support programs for dealers in Europe and the United Kingdom.

Alpha Micro is also active in the original equipment manufacturer (OEM) marketplace. OEMs may purchase components or systems in volume for incorporation into their own products or as part of a private-label program.

Although many companies have entered the multiuser market since 1977, Alpha Micro continues to be recognized as a leader of multiuser micro technology.

The Alpha Micro Organization

Alpha Micro is organized along divisional lines to provide optimum definition and focus of the company's activities. Each operating division is summarized below.

The Computer Systems Division is responsible for planning, marketing, and distributing Alpha Micro's business computer product lines. This division has 10 years of experience in the small business marketplace, in research and development, and in product testing.

The Video Technology Division is responsible for planning, marketing, and distributing Alpha Micro's patented video technology in the form of data backup, archive, and distribution products. These products represent a unique combination of computer and video technologies, offering effective, low-cost solutions to a variety of data storage and distribution challenges.

The Advanced Products Division is responsible for hardware and software development, vendor software analysis, software quality assurance, long-range product planning, technical publications, and evaluating the latest technological advances for possible inclusion in future Alpha Micro products.

The Operations Division manufactures, assembles, and tests all Alpha Micro products. One of this division's most important functions involves quality assurance, working closely with the entire company to maintain high standards. Part of this process is the on-site vendor testing program, which certifies vendor products both prior to and after shipment to Alpha Micro. This is just one example of the comprehensive cycle of testing and quality control that every Alpha Micro system must undergo.

Alpha Micro's Service Division provides a comprehensive package of service and support alternatives available through a worldwide network of field service affiliates and through direct service.

Service and support programs for VARs, OEMs, and end users include troubleshooting by telephone, coordinated worldwide parts repair and delivery service, a variety of software support programs, and a 24-hour interactive communications network, which allows users to view or print out technical information at their location.

This division also maintains a regular curriculum of training classes covering a broad variety of topics. Classes are conducted at Alpha Micro headquarters, regional offices, and customer sites. Most of the more popular courses are available on video tape as well.

PRODUCT PHILOSOPHY

Alpha Micro's product philosophy is based on the idea of providing solutions for business and professional users. The Alpha Micro product line, in its many different configurations, is designed to deliver those solutions.

Our first products were the AM-100, a 16-bit, S-100 bus compatible central processing unit, and the AMOS multiuser, multitasking, time-sharing operating system. They were designed as a bridge between single-user microcomputers and expensive multiuser minicomputers. This challenge required an entirely new approach to microcomputer systems design, emphasizing expandability, ease of use, and price/performance.

Since the introduction of the AM-100 and AMOS, the company has constantly enhanced and expanded its product line, including a migration to the 16/32-bit MC68000/MC68010 and 32-bit MC68020 microprocessors and VME bus architecture.

This progression reflects our basic philosophy of providing computing solutions. Specifically, Alpha Micro has implemented this philosophy in a number of ways: by providing a wide range of cost-effective product choices; by incorporating and improving upon the latest technology; and, perhaps most importantly, by designing systems that can expand to meet growing and changing needs.

The idea of choice—giving our users many options—is reflected in the wide variety of computer languages, communications options, data backup methods, and storage media offered by Alpha Micro. Software solutions are equally varied.

Providing innovative solutions also means merging our unique technological developments with technological breakthroughs developed by others. This challenge has been successfully met throughout the company's history, most recently by combining Alpha Micro computer technology with existing video technology to provide a number of new, cost-effective solutions to data backup, archive, and distribution problems.

Finally, Alpha Micro has been committed since its founding to bringing power and performance together in a computing environment that stresses compatibility. Alpha Micro's product line allows growth from 1 to over 240 users without expensive reprogramming, time-consuming conversions, and costly retraining. The company has also provided numerous bridges—high-level languages, data bases, and specialized interface products—to ease the migration of applications written for other operating systems.

COMPUTER SYSTEM PRODUCT LINE OVERVIEW

AMOS Operating System

Alpha Micro's AMOS operating system was the first multiuser, multitasking operating system developed for a microcomputer. Because it was developed specifically for the workplace, AMOS was designed to give as much of the machine's underlying power to the user as possible, rather than consuming that power itself. This additional power results in high performance in areas such as interactive terminal handling, disk I/O, and job scheduling.

AMOS was also designed to give the software developer maximum flexibility. By providing a relatively small set of rules, AMOS provides freedom and capabilities not available under other operating systems.

Additional features of the AMOS operating system are highlighted below,

Efficiency

For a three-user system, AMOS occupies as little as 256Kb of main memory, whereas other operating systems can require upwards of 1Mb to run efficiently. AMOS also consumes less than 1% of the processor power while allocating time for each user; many other operating systems use 10% to 15%. More processor power provides several advantages, including faster job execution, faster disk access, and support for more users. On the same type of hardware, for example, AMOS can support 40 users, whereas other operating systems may only be able to support 3 to 5.

Utility Programs

AMOS includes a complete library of operating system routines, all written in assembly language. This makes the software developer's job easier without penalizing the user with reduced performance. Also provided is a versatile set of over 150 labor-saving utility programs and routines. In addition, AMOS supports random, sequential, and indexed sequential file access. A text editor and text formatter are included as well.

Commands

New AMOS commands can be written, and existing commands can be changed, renamed, and deleted. Command file processing is supported as well. System capabilities can be expanded with user-developed assembly language programs and with software enhancements developed by Alpha Micro.

Software

AMOS supports one of the largest and best selections of multiusers business solutions available today. Because all of Alpha Micro's systems use AMOS, these systems can use the same applications software. As a business expands, data and programs can be transferred to a new and larger system without expensive reprogramming, time-consuming conversions, and costly retraining.

Languages/Development Tools

Languages supported by AMOS are AlphaBASIC (our enhanced version of BASIC), SMC BASIC, IEEE FORTRAN 77, Pascal, COBOL, C, and a macro-assembler. Alpha Micro also offers AlphaBASE and UNIFY, data base management systems.

Communications

Communications options include AlphaMATE, which allows an IBM PC, XT, AT, or most compatibles to operate as an Alpha Micro terminal. MS-DOS files can be stored on the larger Alpha Micro disk and shared between MS-DOS systems. AlphaMATE also supports phone-line communications between remote MS-DOS systems and a multiuser Alpha Micro system. Another option, AlphaRJE, allows Alpha Micro systems to communicate with each other and with most mainframes that use IBM's 2780/3780 protocol. AlphaRJE is compatible with IBM's job entry system, JES III. Alpha Micro also offers AlphaNET, a video-based local area network that can connect up to 64 Alpha Micro computers.

Menu-Driven

A menu is included with AMOS that guides the user step by step through system functions and applications. System operation can be tailored by custom designing menus and defining which system functions and commands users will be able to perform.

Task Manager

The AMOS Task Manager is a group of programs that can perform tasks unattended, without tying up terminals. Users can schedule jobs to run automatically on a certain day and at a specific time, such as copying data to a video cassette tape every evening at 11 P.M.

Security

AMOS coordinates shared file access and processing to prevent information from being accessed while it is being updated by another user. Further system and file protection is provided by requiring each user to enter a valid account number to log into the system. Password account protection is also supported.

Alpha Micro Computer

Alpha Micro offers a full line of systems based on the MC68000, MC68010, and MC68020 microprocessors and running under our proprietary multiuser, multitasking operating system, AMOS.

Because each system runs under the AMOS operating system, there is software compatibility across the entire line. Alpha Micro's family of systems is described below.

AM-1200 Series

The AM-1200 series of desktop systems is designed to meet the needs of small, growing businesses. The AM-1200 series features the following:

- Supports 1 to 13 users.
- 1 to 4Mb memory
- 20 to 70Mb Winchester hard disk
- One parallel printer port
- Video cassette interface for VCR backup

Also available with this series are streamer tape and floppy disk.

AM-1500 Series

Alpha Micro's state-of-the-art AM-1500 series offers superior performance, expandability, and styling. This series is based on the 16-bit MC68010 microprocessor (can be upgraded to 32-bit 68020) and VME bus architecture. Because the AM-1500 series also runs under AMOS, it is compatible with the entire product line. The AM-1500 series includes the following:

- Supports up to 120 users
- 1 to 16Mb of memory
- 70 to 300Mb high-speed Winchester disk

Also available with this series are a 5¼-inch floppy disk, streamer tape, and VCR backup.

AM-2000 Series

The most powerful of the Alpha Micro VME-bus business systems, the AM-2000 series of micro mainframes, offers the power and user support that a large office needs. This series features the following:

- Supports up to 240 users
- 1 to 36Mb memory
- 70Mb to 1.6Gb high-speed Winchester disk

Options include intelligent serial controllers, math coprocessors, hardware cache memory, and disk accelerators. Backup options include VCR backup, streamer tape, floppy disk, and magnetic tape.

PC Upgrade Package

The PC Upgrade Package converts an IBM PC/XT, AT, or most compatibles into a multiuser system running under AMOS. This package consists of the following:

- IBM PC-size board with MC68000 processor
- 128Kb memory (expandable to 1Mb). Memory is accessible by AMOS or MS-DOS (subject to the limits of MS-DOS).
- AMOS operating system

AMOS and MS-DOS files can be transferred between the two operating systems with supplied utilities. AMOS can use MS-DOS text files directly and vice versa. In addition, utility programs can be written to allow AMOS data files to be used by MS-DOS and vice versa.

Alpha Micro terminals or other standard RS-232 terminals may be used. Both serial and parallel printers are supported with standard serial and parallel I/O boards.

Terminals

AM-62 Monochrome Terminal

The AM-62 is an ergonomically designed terminal featuring a user-selectable 80/132 column display, programmable function keys, line-drawing characters, direct printing, and Televideo 925 and Lear Siegler ADM5 terminal emulations. A long-line RS-422 port for high-speed data transmission is included.

AM-70 Color Terminal

The AM-70 color terminal combines a high-resolution display with eight different colors: red, green, yellow, blue, cyan, magenta, white, and black. Programmable function keys and line-drawing characters are also included.

Printers

Alpha Micro systems support a wide variety of printers.

VIDEO TECHNOLOGY PRODUCT LINE OVERVIEW

In 1981, after 2 years of concentrated development, Alpha Micro introduced technology allowing a VCR to be used, without modification, for backup and archival storage when interfaced to an Alpha Micro computer system. From this product evolved a patented technology with virtually limitless potential, ranging from computer backup to satellite-based networks. Video-based products are discussed below.

Video Cassette Recorder Controller Board

Current video applications are based on the video cassette recorder controller board, which converts digital computer data to an analog video signal, and vice versa. In this way, information can be transferred back and forth between a computer's hard disk and a video device, such as a VCR.

The video controller board includes an on-board CPU, buffers, direct memory access (DMA) data channel, analog-to-digital and digital-to-analog converters, and associated control logic. To the computer, the video board "looks like" an ordinary low-speed disk controller, and the attached video device "thinks" it is receiving a standard video signal from a TV camera.

A combination of firmware (PROM resident on the video board) and software utilities controls the selection and transfer of data, media verification, data verification, error reporting, and other associated functions.

Video Data Backup/Storage

Video backup uses a standard, unmodified VCR to back up a Winchester disk drive to video cassette tape. Backup can be programmed to take place automatically at a predetermined time each day without an operator present. When the VCR is not functioning as a computer peripheral, it can still be used to record and play video tapes.

The media used by this system—low cost, high-capacity (up to 100Mb on a 2-hour tape) video cassette tape—has many advantages. Tapes can be stored easily and safely as backup copies, as well as being used for transferring information to other systems equipped with video backup. Alpha Micro currently distributes the majority of its software on video cassette tape. Tapes can also be used as a system boot device, replacing less convenient media.

Constant product refinement, including the utilization of VLSI technology, has made video backup more reliable than streamer and floppy backup and more reliable than the Winchester disk it is backing up. A number of design features contribute to this high reliability, including a conservative recording technique, a carefully determined data transfer rate, recording redundant copies of the data, using cyclic redundancy checks, and automatically generating error statistics.

In addition, Alpha Micro's video backup software automatically monitors tape and drive quality, providing a reliability ratio of the complete backup system each time a tape is used. If any part of the system starts to degrade, the user is alerted before a problem occurs. This capability is unique to Alpha Micro's video backup.

Video backup is currently in use on more than 19,000 Alpha Micro computers, is available for the IBM PC family and PC compatible systems under the name of Videotrax, and is under development for use with other popular personal computers and operating systems.

Data Broadcasting

With video-based data broadcasting, data can be sent anywhere that a TV picture can be received. For example, a company's computer data can be converted into standard television video signals and, using existing broadcasting facilities, transmitted to an orbiting satellite. The satellite then relays the data back down to each of the company's locations which receive

the signals simultaneously. Computers at each location reconvert the signals so that information can be stored on the computer or a VCR tape for downloading. Confidentiality may be ensured by a variety of hardware- and software-encoding techniques. Data and full-motion video/audio may be combined in the same broadcast.

Data may also be sent via any existing cable or microwave system. Because the cost of dishes, broadcasting facilities, and actual transmission time is rapidly decreasing, data broadcasting is especially well suited for businesses, government agencies, and educational institutions.

Video Wide Area Network

Data broadcasting (as described above) is essentially unidirectional. If receiving sites also have the ability to transmit, then the system becomes a bidirectional wide area network, allowing any site to send data to any other site. Under certain conditions, normal dial-up switched telephone lines can be used instead of broadcasting facilities.

Other Video Products

Other applications for video technology range from mainframes to home computers. For example, video technology could be used to interface with currently emerging high-capacity laser disks, or as part of a back-end main-frame video controller. This technology could also be used to provide on-line tutorials with video and voice signals appearing on one monitor while data are sent simultaneously to the second monitor. Video technology is proven and reliable. The potential applications are limited only by the imagination.

THE FUTURE

Alpha Micro's continued commitment to technology remains the foundation of the company. While accepting the importance of external developments and compatibility, the company concentrates on innovations that set it apart and make it unique.

There are at least two areas in which Alpha Micro is making extensive application of company-created technologies. The first is our proprietary video technology. This technology, which was developed exclusively for use with Alpha Micro systems, is now being developed into a family of products with OEM and consumer applications throughout the computer industry. This is a prime example of bringing Alpha Micro technology to other markets based on unique, cost-effective applications.

A second major area is Alpha Micro's operating system, AMOS. AMOS continues to be respected for its efficiency and performance. Years of enhancements have been incorporated since its debut as the first micro-processor-based, multiuser operating system 10 years ago.

New Alpha Micro products, whether hardware or software, will continue to meet the same exacting standards of quality and functionality that have been used to measure all previous Alpha Micro developments. Alpha Micro will continue to bring power and performance together in a computing environment that stresses compatibility, as well as the intelligent integration of our

own technological breakthroughs with those developed by others. Future Alpha Micro products will continue to reflect a design and engineering philosophy that is sensitive to our customer's needs.

PEGGY McCRANIE

ALTOS COMPUTER SYSTEMS

THE MARKET

During the course of the computer market's dramatic evolution, a persistent gap developed in the range of machines offered to the computing public.

For large companies with hundreds of thousands or millions of dollars to spend on data processing, the mainframe was the only game in town. For somewhat smaller companies or departments of larger companies with hundreds of thousands of dollars in data processing budgets, the minicomputer was the solution of choice. And the personal computer could certainly satisfy the needs of the single user. Yet between the single user and the smaller company/division or department, a gap existed that could not be filled with existing products. Stand-alone micros were too limited and minis were too expensive to meet the requirements of the small business with several users and a data processing budget in the tens of thousands of dollars.

Enter the multiuser microsystem (MUMS), a micro-based computer that could service the computing needs of many users in a smaller company or department, yet was much less expensive than a minicomputer.

Among the companies to pioneer development of MUMS was Altos Computer Systems. Altos realized that the economics of data processing for small companies would place a tremendous emphasis on hardware price/performance characteristics and on the advantages of being able to amortize the costs of hardware and software over several users. So the company began developing a line of multiuser systems based on commercially standard microprocessors that would give users the most power available for their money.

In the late 1970s, the gap between minis and stand-alone micros became a commercially viable market niche, and Altos was soon joined by Northstar, Televideo, Fortune, and others trying to exploit the possibilities of low-cost multiuser computing.

Altos' product line expanded rapidly in response to a substantial pent-up demand for its systems. Starting with three- and four-user systems, the company was soon offering both stand-alone machines and also micro-based systems that could handle 1 to 20 and 30 users—formerly the sole province of minicomputer makers like IBM, DEC, and NCR.

InfoCorp, a market research firm that tracks the computer industry, has projected that the worldwide market for multiuser systems will grow from \$27.7 billion in 1987 to \$43.5 billion in 1990.

CORPORATE OVERVIEW

Founded in 1977 by British entrepreneur David Jackson, Altos Computer Systems has steadily evolved into a company relying more on marketing savvy, technological foresight, and management teamwork than on entrepreneurial

flair. Now, 10 years after its inception, Altos is pursuing a three-pronged strategy of technological innovation, vertical market penetration, and balanced business growth, worldwide.

Technologically, Altos has been able to outcompete industry giants like IBM, AT&T, and DEC to deliver a long list of marker "first," such as

- 8-bit MUMS
- 16-bit XENIX MUMS
- 32-bit MC68020 UNIX V MUMS
- UNIX/XENIX network (WorkNet)
- PC DOS/XENIX network (PC Path)

The primary targets for Altos' marketing strategy are (a) businesses with revenues between \$1 million and \$25 million, (b) government entities, and (c) departments in Fortune 1000 companies. To penetrate the vertical markets in categories a and b above, the company relies on a network of resellers expert in the requirements of those markets. Altos sells machines to these resellers and systems integrators. The resellers then configure Altos systems—including customized applications software—specifically to meet the demands of the markets in which they specialize.

Altos' expansion plans do not include the "growth at all costs" strategy pursued by many microsystems companies, some of whom have discovered that the costs are too high for survival. Instead, Altos has followed a steady upward growth curve to \$145 million in fiscal 1987. The company has been profitable every quarter during 10 years in a market where boom-and-bust cycles are accepted as a fact of life.

In San Jose, California, Altos has more than 90,000 square feet of production area, where it assembles components, parts, and subassemblies purchased from others. Components and printed circuit boards are subjected to a battery of test and inspections before assembly.

The company is publicly owned and is traded in the NASDAQ over-the-counter stock market (symbol "ALTO"). The initial public offering in November of 1982 raised \$59 million from 3,630,000 shares. At the end of fiscal 1986, the company had more than \$50 million in cash reserves and virtually no long-term debt.

Headquartered in San Jose, California—with subsidiaries in Toronto, Sydney, Paris, Munich, Hamburg, London, Copenhagen, and Milan—the company employs over 700 people and now has an installed base of 85,000 systems in more than 57 countries worldwide.

HARDWARE OVERVIEW

Altos' MUMS can be divided into two families. One is based on two Intel chips: the 80286 and the 80386. The second family is based on the Motorola 68020 chip. All Altos systems share one characteristic: They use a single, powerful CPU (central processing unit) and terminals to allow users to share applications, data, and peripherals. In this system, the CPU controls system functions, schedules disk and peripheral activities, and sets processing priorities in addition to performing the processing required by each user. Coprocessors then handle input/output and other functions.

The hardware is designed to allow users to choose from a number of microprocessors, disk storage options, streaming tape drives, and random access memory (RAM) capacities. As users' requirements grow, their Altos system can grow as well.

Altos systems are designed with upward compatibility in mind. Should users outgrow one system, they can upgrade to a larger one and not risk their software investment. More details on Altos hardware follow, including Table 1 which gives the specifications of the present line of computer models.

16-Bit Business Systems

Altos' wide range of 16-bit multiuser, multitasking products are competitive in price and performance with minicomputers and are designed to perform larger, more comprehensive tasks for more users at a more economical cost per user than individual personal computers.

Altos 886 Series

The 886 series utilizes the Intel 80286 CPU and is designed to accommodate up to eight users. It is configured with 1Mb of RAM, 25, 50, or 80Mb (unformatted) hard disk, and a 60Mb streaming magnetic tape cartridge.

Altos 1086/2086 Series

The 1086/2086 series are floor-standing supermicrocomputers utilizing the Intel 80286 CPU. They feature modular expansion capabilities for adding disk drives and communications boards and accommodate up to 20 users. Each system includes a PC/AT compatible diskette drive, 50 or 80Mb (unformatted) hard disk, 60Mb streaming tape unit, parallel printer port, 1, 2, or 4Mb RAM, and runs the XENIX 3.0 operating system.

Altos 386 Series

The Altos 386 Series are floor-standing supermicrocomputers utilizing the Intel 80386 CPU. They feature a variety of expansion capabilities to add storage and communications capabilities, accommodate up to 64 users, and run the XENIX 5 operating system.

Altos 3068

The 3068 is a floor-standing supermicrocomputer utilizing the 12.5MHz Motorola 68020 CPU. It features modular expansion capabilities for adding disk drives, additional memory, and communications and proprietary boards and accommodates up to 30 users. Remote diagnostics are available as an option. Each system includes 1Mb RAM, expandable to 16Mb, a 20Mb hard disk, and a 1.2Mb floppy disk drive. Hard disks can be upgraded to 240Mb (unformatted) in 20, 60, or 80Mb increments. A streaming magnetic tape unit with up to 60Mb of storage is also available. The 3068 runs the UNIX V operating system.

Table 1 Altos Hardware Systems

	Intel			Series 386	Motorola 3068
	886	1086	2086		
Microprocessor	16-bit Intel 80286	16-bit Intel 80286	16-bit Intel 80286	32-bit Intel 80386	32-bit Motorola 68020
RAM capacity	1-2Mb	1-8Mb	2-8Mb	2-16Mb	1-16Mb
Floppy disk capacity (unformatted)	1-1.16Mb	1-1.6Mb	1-1.6Mb	1-1.6Mb	720Kb-1.2Mb
Hard disk storage (unformatted)	25-65Mb	50-240Mb	80-240Mb	80Mb-1.2Gb	40-80Mb
Streaming magnetic tape backup	886-T 60Mb	60Mb	60Mb	60 or 125Mb	60Mb
Number of users	1-8	1-10	1-20	1-64	1-30
Operating system	XENIX 3.0	XENIX 3.0	XENIX 3.0	XENIX 5	UNIX V
Date of introduction	October 1985	October 1985	June 1985	May 1987	March 1985

Terminals

Altos III

The Altos III features an ergonomically designed monitor a 14-inch, nonglare screen, a detachable keyboard with 32 programmable function keys, and an auxiliary printer port that allows data transfer at rates between 110 and 19.2Kbaud. It has ASCII and graphics character sets and is fully compatible with DEC VT-100 terminals. It can also emulate the Televideo TV1-901 terminal, which allows it to attach 80-bit systems.

Altos IV

The Altos IV is an entry-level ASCII-based terminal featuring a 14-inch screen, an 80-column display, and a detachable keyboard with a numeric keypad and 16 function keys, which can control 32 functions. The Altos IV is compatible with a wide range of terminal offerings.

SOFTWARE OVERVIEW

Altos offers a wide range of commercially available systems and applications software.

The Intel-based systems use XENIX (developed jointly by Microsoft and Altos) and the Motorola-based systems use Bell Labs' UNIX System V and PICK.

The applications software Altos offers includes accounting, office automation, data base management, and other horizontal packages. The company also actively pursues third-party development agreements with a number of software vendors specializing in various vertical markets.

Altos also offers communications and networking software. More details on Altos software can be found in the following pages.

Operating Systems

Altos configures its system with commercially available operating systems software, rather than with a proprietary operating system, to provide users ready access to a wide variety of applications programs developed by independent vendors. By using compatible operating systems for 16-bit micro-computer products, the company protects the software investment of its customers, allowing them to transport their software and most applications packages from a 16- to a 32-bit system. Operating systems available are concurrent DOS, XENIX, UNIX, PICK, and THEOS.

For its Intel-based systems, Altos offers XENIX 3.0 and XENIX 5, an enhanced version of AT&T Bell Laboratories' UNIX operating system. XENIX was developed jointly by Microsoft Corporation and Altos to provide relatively unsophisticated users with a powerful and flexible operating system.

UNIX System V, the most recent version of UNIX, released by AT&T Bell Laboratories for commercial use, is the primary operating system for the 3068 supermicrocomputers.

Computer Languages

A variety of languages are supported for each system. The list includes several versions of BASIC and COBOL as well as Pascal, FORTRAN, and C.

Applications Software

Altos relies primarily on applications packages developed and sold by independent vendors for specific vertical markets. However, the company has also developed its own software to meet the business and office automation needs of its customers.

A modularized accounting system called Altos Accountant handles accounts payable, accounts receivable, general ledger, inventory, job costing, payroll, and sales order processing. Also developed by Altos is a sophisticated spreadsheet package that includes Microsoft's Multiplan. For word processing, Altos offers Uniplex.

Altos Office Manager (AOM II Plus) is an integrated office automation software package for Altos multiuser systems that includes word processing, financial modeling, business graphics, data base management, communications, calendar, and electronic mail. AOM's unique menu system and the AOM Tool Kit allow resellers to integrate AOM's horizontal applications with their own vertical market software to provide a comprehensive, off-the-shelf office solution.

Communications and Network Software

Several communications software packages that allow file exchange with other computers are available to Altos systems users.

WorkNet, Altos' networking software, permits users to link together multiple 16-bit Altos microcomputers into a local area network (LAN). WorkNet is entirely transparent to users and provides electronic mail services, remote disk access, file transfer, remote terminal access, and communications gateways to all workstations on the network.

PC Path allows IBM PCs and PC compatibles running PC-DOS applications to share disks and printers on WorkNet. This allows PCs to store files on Altos server units and spool printouts to shared network printers. Users can access XENIX applications directly from the PCs, including a variety of communications gateways.

PC Plex

PC Plex software allows IBM PCs and compatibles to extract information from data bases on Altos multiuser systems for manipulation in PC files. Altos PC Plex automatically formats the data for PC applications such as Lotus 1-2-3, dBASE, WordStar, Multiplan, and PC graphics packages. PC Plex also provides file transfer, network file management, and terminal emulation for the PC user.

DISTRIBUTION

Altos uses a distribution strategy that comprises essentially four channels.

The first channel is the direct sale of systems to original equipment manufacturers (OEMs), who then sell reconfigured systems to Fortune 1000 companies.

The second channel is the direct sale of systems to volume resellers (large value-added resellers [VARs]), who then tailor the systems to the needs of medium-sized businesses or particular vertical markets.

The third channel is the sale of systems to distributors, who then sell to 1,200 dealers.

The fourth channel is systems integrators who purchase products from Altos and add a variety of components to provide complete office systems to agencies of the federal government.

DOMESTIC ENVIRONMENT

For its reseller distribution channel and Intel product family, Altos' primary market is small businesses with revenues between \$1 million and \$25 million. Within this general market, Altos has targeted several vertical markets with high growth potential, including medical, dental, construction, accounting, manufacturing, and wholesale distribution.

The Motorola product family and higher-end Intel systems make their way to market through the distribution channel made up of major accounts, OEMs, and larger VARs. The markets served by this channel include departments of Fortune 1000 companies; federal, state, and local governments; and vertically oriented businesses.

Altos competes with AT&T, Unisys, Convergent, DEC, IBM, and NCR. To compete successfully, the company relies on three key technological characteristics: the best price/performance ratio in the 2- to 64-user environments; system expandability and compatibility; and availability of the standard UNIX/XENIX multiuser and multitasking operating system.

The company also relies on three key business accomplishments: wide acceptance as the small business system; distribution channel strength due to market longevity; and strong financial reserves and corporate stability.

INTERNATIONAL ENVIRONMENT

In Western Europe, Altos has established a strong position in the MUMS markets using a distribution strategy similar to its channels in the United States. The company has sales offices in Munich, Hamburg, Paris, Copenhagen, Milan, and London.

Altos is strengthening its competitive position in Asia, Canada, Australia, and Latin America and has recently signed contracts with the People's Republic of China, India, and Korea.

The company generates about 45% of its business from international sales.

BUSINESS PARTNERSHIPS

A significant piece of Altos' strategy for balanced and profitable growth is its ability to form strategic alliances with partners whose strengths complement Altos' own. These relationships range from equity participation to mutually beneficial cooperation, including

- Ownership or investment in software companies specializing in a variety of products that complement Altos software families.
- Ongoing development of the company's network of independent distributors and dealers in 65 countries.

RESELLER SUPPORT

Altos has 19 programs in place, designed specifically to benefit its resellers in areas ranging from technical education to marketing and presales support, vertical market consulting, co-op advertising, and software support.

Altos Education Center

This program was founded in 1981 to teach dealers, distributors, resellers, and OEMs how to repair and support Altos systems. At that time, it had one instructor.

The center now has its own building, a program of 200 seminars per year, and 13 instructors in several locations nationwide. The seminars now cover how to use Altos systems, how to position them competitively, how to market and sell them, as well as how to repair them.

Altos Software Availability Program (ASAP)

This program is a mechanism for encouraging the development of third-party software for Altos machines and getting it into the various Altos distribution channels. The program includes regular software education seminars and product demonstrations, as well as test and evaluation of third-party ASAP software to run on Altos systems.

Ask Altos Hotline

Distinct from the technical support hotline, Ask Altos is a means of providing resellers with presales support and information on product specifications, Education Center classes, ASAP courses, spare parts, and competitive products.

Technical Support

Altos provides technical support to resellers via a toll-free hotline to headquarters in San Jose, California, as well as field support and authorization of Altos resellers as service centers.

MIKE SANCHEZ

AMERICAN SOCIETY FOR INFORMATION SCIENCE—HISTORY

INTRODUCTION

The American Society for Information Science (ASIS), known as the American Documentation Institute (ADI) from 1937 to 1968, is a "nonprofit professional association organized for scientific, literary, and educational purposes and dedicated to the creation, organization, dissemination, and application of knowledge relating to information and its transfer" [1]. The primary purpose of ASIS is to develop advanced methods and techniques that contribute to the more efficient use of information.

The American Documentation Institute concentrated on microphotographic documentation. In 1947 ADI joined the International Federation for Documentation. Individuals showed such an interest in documentation that ADI, formerly consisting of representatives of nominating agencies, opened the membership to individuals in 1952. Services and programs of ADI were enlarged and expanded to meet the needs of the members in the 1960s and 1970s. In January 1968 the American Documentation Institute became the American Society for Information Science, reflecting the change in emphasis from documentation to information science.

Appendix 1 provides a list of ASIS presidents, and significant events in its history are outlined in Appendix 2.

ORIGIN

Watson Davis provided the impetus for development of ADI through his efforts as a leader in Science Service, an institution whose sole aim is the popularization of science. His work with documentation techniques led to the formation of Bibliofilm Service and Auxiliary Publication Service. The interest of librarians, editors, and others in microphotography and the use of microfilm resulted, in part, from Science Service activity.

Documentation efforts in Science Service commenced in 1926 when Watson Davis and Dr. Edwin E. Slosson, first director of Science Service, envisioned the potential usefulness of microfilm for documentation. A mimeographed memorandum, "Project for Scientific Publication and Bibliography," by Watson Davis was issued in August 1933. This memorandum described a possible central scientific information institute (SII), methods of publication, and bibliography and suggested the advantages of microphotographic duplication and mechanization of bibliography methods [2].

Science Service further pioneered efforts in documentation by holding conferences where proposals for the use of microphotographic duplication were discussed. Among those attending a conference on the subject of film copying of documents on November 5, 1934, were three documentalists: Dr.

R. H. Draeger, Medical Department of the United States Navy; Miss Claribel R. Barnett, librarian, United States Department of Agriculture; and Atherton Seidell, National Institute of Health.

As a result of this conference, Bibliofilm Service officially began on November 15, 1934. This service was a nonprofit, cooperative service:

started on enthusiasm, cooperation and hope, for no special funds for the service were available. The films and necessary chemicals and equipment were furnished by Dr. Seidell and Dr. Draeger. The Library supplied the space for carrying on the work and one Library assistant [3].

Bibliofilm Service, primarily limited to materials in the Library of the United States Department of Agriculture, provided inexpensive microfilm and photoprint copies of scientific articles in order to decrease interlibrary loans of books and extend the resources of the library to remote areas. Watson Davis envisioned Bibliofilm Service as a network of "libraries cooperating to make their holdings usable by one another, so that the library client had access to all by submitting his request to one" [4]. Copyright developed as an issue from the initial stages of operation.

It soon became desirable to establish a cooperative agreement between the Library of the United States Department of Agriculture and Science Service.

Under the cooperative arrangement, which became effective January 1, 1936, Science Service has taken over the Bibliofilm Service, including the business management, all the routine photographic work, and the experimental work of developing mechanisms [5].

However, the photographic laboratory remained in the Library of the United States Department of Agriculture, acting as the agent for the individual or institution ordering the microfilm copies.

By mid-1935 it was possible to initiate some of the phases of the "Project for Scientific Publication and Bibliography" through cooperation with other institutions. A Chemical Foundation grant of \$15,000 allowed Science Service to organize its Documentation Division in July 1935 for "initial exploration, development of mechanisms and inauguration of some phases of the publication project" [6]. Prior to this, the proposed activities of the Documentation Division were designated to the tentatively named "Scientific Information Institute." This grant provided funds for the operation of Bibliofilm Service and the Auxiliary Publication Service through March 1937.

Progress in the use of microfilms was dependent upon satisfactory equipment for reading the film. The necessary reading devices, cameras, and other mechanisms were designed and constructed in model form as a result of cooperative efforts under the auspices of the Documentation Division of Science Service. The mechanism development was terminated in June 1936 when these mechanisms became commercially available.

The Documentation Division of Science Service established the Auxiliary Publication Service on March 1, 1936.

The plan of Auxiliary Publication...was proposed in a memo issued June 20, 1935, "Project for Publication of Scientific Papers and Mono-

graphs That Can Not Now Secure Prompt and Complete Issuance," by Watson Davis. It was...discussed and presented in detail at a conference on documentation in Washington called by Science Service on June 30, 1935 [7].

The Auxiliary Publication Service provided scientists and scholars prompt access, in a convenient and usable form, to important papers too long to be published in the journals. An abstract or notice of the original paper appeared in the appropriate journal, together with a statement that the entire paper, with its charts, diagrams, or illustration, if any, could be obtained as microfilms or photoprints from Science Service. The documents, available for a nominal fee, were always "in print" as the negative could be used to make a copy at any time. Reproduction was done in the laboratory of the Library of the United States Department of Agriculture, where Bibliofilm Service was located. In the Auxiliary Publication Service, microfilm copies supplemented, rather than replaced, other forms of publication.

FOUNDING

Under the leadership of Watson Davis, a group of men interested in development of a documentation association organized a conference in March 1937, which drew approximately 30 representatives from national councils, societies, libraries, and other organizations to Washington, D.C. The American Documentation Institute (ADI) stemmed from this conference as a not-for-profit Delaware corporation for the promotion and development of documentation in scientific and scholarly fields.

At the documentation conference on March 13, 1937, the representatives accepted the tentative Articles of Incorporation, elected a Board of Trustees, and chose Watson Davis to serve as president of the American Documentation Institute. They designed the organizational structure of ADI to

invite the learned and scientific societies of the United States each to appoint a single representative who will be a member of the American Documentation Institute, and those members, meeting annually, will elect the trustees who become the executive and administrative department of the organization [8].

There were no dues, no costs, and no commitments from the respective organizations, which were granted memberships for a period of 3 years. The members attended the annual meetings, where the main topic of business was to elect the Board of Trustees. The trustees, also members of the corporation, governed ADI by appointing the officers of the corporation and managing the property, activities, and any other business of the institute.

Bibliofilm Service and Auxiliary Publication Service, formerly conducted by the Documentation Division of Science Service, became the nucleus of ADI. These services were transferred to ADI on July 1, 1937, for further development and expansion.

Bibliofilm Service continued to operate in the Library of the United States Department of Agriculture, where ADI also maintained microphoto-

graphic laboratories. In fall 1937 Bibliofilm Service expanded to include the Library of Congress and the Army Medical Library. It extended in late 1938 to the Geological Survey and the United States Department of the Interior, thus forming the first general microfilm service to cover collections of the federal libraries in Washington, D.C. This service acted as

a national clearinghouse for copying orders for research materials, fitting them through its own and other services, for materials located in substantially all Washington and Baltimore libraries, bureaus and institutions and in other cities here and abroad [9].

The design and development of microphotographic mechanisms, a task of ADI, was accomplished through Science Service with the cooperation of the Chemical Foundation, United States Navy, Bureau of the Census, Works Progress Administration, and the Library of Congress. Approximately \$25,000 went into the development and equipment of a laboratory located in the Library of the United States Department of Agriculture capable of economical and efficient operation of Bibliofilm Service [10]. ADI did not engage in the sale or manufacture of microphotographic mechanisms as it expected commercial enterprise to conduct this work.

The American Documentation Institute continued the Auxiliary Publication Service, thus making material which could not be published in full, as well as older manuscripts, out-of-print materials, and dissertations accessible through the use of microfilm and photoprints. Approximately 30 journals and organizations cooperated, without any investment or subsidy by author or editor, by depositing their materials with ADI.

As one of its first projects, ADI participated in the first World Congress of Universal Documentation, an effort toward world cooperation in documentation, held in Paris, August 1937. Watson Davis chaired the American delegation at this conference. By 1939 the influence of ADI had become international. During this year, Dr. Atherton Seidell initiated a similar organization in France, *L Bibliofilm de la Commission Internationale des Industries Agricoles*. The Bibliofilm Service of ADI filled orders for this organization through correspondence not only with France and other European countries but also with South American and Asian countries [11].

Various foundations recognized the advantages of microfilm and generously contributed to ADI. The Chemical Foundation grant provided funds from 1936 to 1938. An emergency grant from the Rockefeller Foundation assured continuation of services during the early months of 1938. The Carnegie Corporation provided a grant to ADI in 1939 for the hiring of a director, Cuthbert Lee, who had direct charge of the operating activities of ADI.

1940-1949

The American Documentation Institute, after a strong beginning, decreased its activities during the 1940s. The institute continued to copy whole journals either on microfilm or by photoprint, thereby permitting libraries and individuals to obtain missing issues of scientific journals from Europe during this period. From 1941 to 1944 ADI established and maintained an Oriental Science Literature Service. It published the *Far Eastern Science Bulle-*

tin, which contained abstracts of scientific papers published in the Japanese and Chinese languages with adjacent translation facilities [12].

Bibliofilm Service continued its programs with branches in the United States Department of Agriculture, the United States Geological Survey, the National Bureau of Standards, and the Library of Congress. This service located and filmed all types of material. The Library of the United States Department of Agriculture assumed the operation of Bibliofilm Service in 1941.

The American Documentation Institute is continuing to cooperate by loan of equipment and joining in facilitating maximum service. The Institute is enabled by this arrangement to give major attention to the important function of auxiliary publication through microfilm, sets of journals in microfilm, and other activities of service to the scientific and scholarly world [13].

In 1946 ADI published the first edition of the *Catalog of Auxiliary Publications in Microfilms and Photoprints*, a listing of approximately 2,000 documents, as well as microfilmed sets of journals deposited with ADI. These documents, which were available upon request for a nominal fee, covered a wide range of subjects and enabled libraries to obtain rare and out-of-print journals of a scientific and scholarly nature. To facilitate this service, the laboratories of ADI, located in the Library of the United States Department of Agriculture under cooperative agreement, were used jointly for the operation of Bibliofilm Service and the work of ADI, including that of the Auxiliary Publication Service.

The International Federation for Documentation (FID) actively sought an American organization to hold a national membership in the federation. The American Documentation Institute and the American Library Association jointly sponsored a delegate, Vernon Tate, to the 1947 meeting of the FID in Bern, Switzerland. Tate, authorized to apply for American membership on behalf of ADI, found unanimous acceptance of the institute as a national member of the foundation.

1950-1959

The period 1950 to 1959 reflected an increase of interest in ADI. Luther H. Evans, librarian of Congress, was elected president of ADI in 1950. Thus, the Library of Congress became the executive headquarters of ADI for the next 10 years. The institute began to take a new direction.

Perhaps the most significant project of ADI in this decade, achieved through the institute's Committee on Publication, was the publication of the quarterly journal *American Documentation*. Vernon D. Tate served as editor from 1950 to 1952. Historically, *American Documentation* continued the *Journal of Documentary Reproduction*, founded in 1937 by the Committee on Photographic Reproduction of Library Materials of the American Library Association.

The most notable event in the history of ADI is the membership decision, occurring in 1952, that granted individual membership by the institute. On September 26, 1952, the new Constitution and Bylaws of ADI won

approval by a majority vote of the membership. The new structure of the institute established three classes of membership: members designated by nominating agencies, individual members, and institutional members. All of these members paid dues. The membership gained the power to vote and to elect all the officers of ADI, and the powers prescribed for the members resided in the membership in that the membership made recommendations [14]. The new Constitution provided for a Council—consisting of the president, president-elect, past president, treasurer, and two other members—to manage the corporation, rather than having a Board of Trustees. Hence, ADI became the organization for individuals interested in documentation and information services.

Activities of ADI included study of bibliographic control, concern with the problems of cataloging and classification, and the continuation and expansion of ADI's publication program. The Reproduction Service of ADI united the Journal Reproduction and the Auxiliary Publication services. In the early 1950s there had been a drastic decline in revenue from the sale of positive prints of its microfilm of long-run journals. The dwindling of this program, formerly ADI's principal source of income, was due partly to restrictions on filming imposed by copyright considerations but was mostly due to the lack of recent additions, the depletion of the market for prints for those that were available, and the lack of adequate publicity. The Library of Congress, assuming title to all materials collected and deposited through this program, provided maintenance and service for the Reproduction Service. The institute received a small royalty from this [15].

The 1954 annual meeting of ADI proved significant in three respects: The meeting was held away from Washington, D.C., for the first time; papers were volunteered or sponsored by members, rather than a program of solicited papers; and Dr. Atherton Seidell received the first American Documentation Institute Award. An important step toward the advancement of ADI began with the establishment of the first chapter of the institute, the Potomac Valley Chapter, approved on June 26, 1953. This local chapter served as a model for formation of similar groups throughout the country.

Cooperating with other societies, the American Documentation Institute, the Special Libraries Association, and the Association of Technical Writers and Editors sponsored an Institute on Special Librarianship and Documentation on June 8, 1956. An International Conference on Scientific Information, held in November 1958 in Washington, D.C., received support from the American Documentation Institute, the National Academy of Sciences/National Research Council, and the National Science Foundation. This conference served as a forum for a thorough discussion of the status of research on scientific information problems and methods of solving them.

1960–1967

Although the 1950s disclosed a steady growth in membership and activities, it was not until the 1960s that ADI became an extremely active organization with rapidly increasing membership and programs. The establishment of more regional chapters promoted ADI in certain areas of the nation. Membership grew from 300 in 1959 to more than 2,500 in 1967. Accomplishments for ADI included a broadened publications program, innovations at the

annual meetings, and more cooperation with other professional organizations. An initial thrust for these accomplishments may be traced to a National Science Foundation grant totaling \$49,500 received over a 3-year period, 1960 to 1962. The grant, used for the enlargement and improvement of ADI and for the funding of a secretariat, brought new life to ADI.

The establishment of a secretariat with a salaried executive director and offices in Washington, D.C., became a reality in October 1960. As the first full-time executive director, Dr. John B. Kaiser represented ADI professionally and managed the national headquarters. When the National Science Foundation grant expired, the secretariat continued but proved to be a financial burden on the institute.

The American Documentation Institute had never been a self-supporting organization, but it did manage to survive with the aid of grants which ensured operation of the programs and continuation of ADI. By 1964 ADI was close to bankruptcy due to failure to obtain a grant. The Council signed a contract in 1964 with James E. Bryan, Inc., for the management of ADI. This agency helped the institute to develop into a strong, self-sustaining organization.

After critically evaluating ADI's role in the International Federation for Documentation, the institute requested the National Academy of Sciences/National Research Council to consider the United States national membership in the federation. At the 1960 FID conference, ADI asked to be relieved of its national membership to become an associate corporate member, whereas the National Academy of Sciences/National Research Council asked to become the United States national member. These requests were approved and became effective during the conference [16].

The annual meetings of ADI became more sophisticated as the membership grew. Features inaugurated at the 1961 annual meeting included tutorial sessions, specialist seminars, state-of-the-art reports, author forums, and exhibitors' presentations. The Placement Service initiated at the 1962 annual meeting became part of the institute's national program. In 1963 papers circulated in advance of the convening of the meeting as the first book of technical articles every produced by type set automatically by electronic processing equipment. A National Science Foundation grant supported this project.

The 1965 annual meeting of ADI, held in conjunction with the International Federation for Documentation Congress in Washington, D.C., spawned eight Special Interest Groups (SIGs). The Special Interest Groups, professional societies within ADI, gave scope and focus to the divergent interests of the institute's members by providing "an opportunity to identify and meet with colleagues having similar special interests, and to cultivate these interests through conferences, publications and other activities" [17].

The American Documentation Institute sponsored the Working Symposium on Education for Information Science in 1965, which later became known as the Airlie Conference. This symposium created a better perspective concerning the educational aims and policy of information science. A grant from the United States Office of Education helped support this conference, which resulted in publication of the volume *Education for Information Science*.

In 1961 the former "President's Newsletter," published in *American Documentation* since 1951, became the *ADI Newsletter*. Issued sporadically

at first, "its main purpose is to serve as a clearinghouse for news of Council and Committee action and a forum for membership opinion and suggestion" [18]. The "Proceedings of Annual Meetings" appeared in *American Documentation* until 1964 when the *Proceedings* began separate publication. Each volume of the *Proceedings*, published prior to the annual meeting, is dedicated to the theme of the meeting.

In 1965 the National Science Federation awarded a grant of \$60,500 to ADI to initiate a new publication, *Annual Review of Information Science and Technology*. A subcontract between the System Development Corporation and ADI implemented the project whereby the System Development Corporation collaborated with ADI by supplying an editor and staff. Dr. Carlos A. Cuadra served as editor from 1965 to 1975. The first volume of the *Annual Review of Information Science and Technology* reviewed 1965 and some previous work. Each volume in the series covers the literature and developments of the calendar year prior to the year of publication and provides brief summaries of new contributions to the state of the art in documentation and in information science.

Documentation Abstracts began in March 1966 as an expansion of a service previously provided in the "Literature Notes" of *American Documentation* and in the "Annotated Bibliography" of *Chemical Literature*. In June 1966 *Documentation Abstracts* added the "Documentation Digest" of *Sci-Tech News*. Thus, the American Documentation Institute, the Chemical Literature Division of the American Chemical Society, and the Special Libraries Association sponsored *Documentation Abstracts*. This quarterly abstracts journal was a comprehensive source of information about the literature in the field of documentation and related areas.

Other accomplishments during this period in ADI's history included a new Constitution and Bylaws adopted in 1965:

its purpose being to improve and expand the ADI's organic law; to provide for changed needs and functions; to clarify the status of chapters and special interest groups; to simplify and rationalize the ADI committee structure; and to integrate national and chapter committee structures [19].

The institute's "Distinguished Lectureship" provided an opportunity for chapters to hear outstanding authorities in information science speak on topics of current interest. The interest shown by students in the work of ADI developed into Student Chapters. Local chapters of ADI cosponsored the Student Chapters and gave students an opportunity to participate in chapter activities. In October 1964 the Council established a Scholarship Fund for Information Science in memory of Hans Peter Luhn, former president of ADI. The institute presented the annual Award of Merit "to the member of the profession who made the greatest contribution to the field for the year" [20]. Hans Peter Luhn received the first award posthumously in 1964.

By 1967 ADI exhibited characteristics of a professional organization: financial responsibility, a Code of Ethics, regular production of critical reviews of developments, and cooperative projects with peer organizations. To reflect its professional characteristics, ADI considered a new name. In 1937 the interests of the charter members of ADI focused on documents of

all kinds in all fields of human activity. Until 1952 ADI membership consisted of representatives appointed from various institutions. After permitting individual membership in 1952, many members felt that the term "documentation" did not effectively represent the full scope of the interests and activities of ADI. Also, the word "institute" did not clearly indicate a professional society. In June 1967 "the Council of the American Documentation Institute voted unanimously to recommend to the membership that the American Documentation Institute become the 'American Society for Information Science,' effective January 1, 1968" [21].

1968–1973

The American Documentation Institute became the American Society for Information Science (ASIS) on January 1, 1968, after more than 80% of the voting members favored the proposal. An amendment to the Constitution in September 1968 permitted a change in emphasis from documentation to information science. In part, the amendment defined ASIS as

a non-profit professional association organized for scientific, literary and educational purposes and dedicated to the creation, organization, dissemination and application of knowledge concerning information and its transfer. The membership is drawn from all fields in which information is important either as an object of study or as a means to an end [22].

During this period, the financial status of ASIS improved, giving stability to the organization. In the late 1960s the society maintained fiscal balance primarily through journal subscriptions and income from annual meetings. By late 1973 a full-scale publications program of ASIS had grown to the point of generating enough income to enable ASIS to remain on a sound financial basis.

In fall 1969 the Council decided to establish its own administrative mechanism because of the broadening interests and activities of ASIS. After terminating the management contract with James E. Bryan, Inc., ASIS established an independent administrative office and staff for the society in Washington, D.C. In April 1970 Herbert R. Koller assumed the duties of executive director. Upon the resignation of Koller, Joshua I. Smith became the executive director of ASIS in October 1973.

In late 1968 Herbert White, president of ASIS, and Joseph Becker, president of the Special Libraries Association, explored the possibility of closer relationships between the two organizations. The initial point of discussion involved the possibility of a joint headquarters operation, but merger of the organizations quickly developed as a logical outcome. The objectives, programs, and memberships of the Special Libraries Association and the American Society for Information Science overlapped to some extent. However, the society's approach to the analysis and use of information was theoretical and research oriented, whereas the Special Libraries Association's approach was practical [23].

A committee composed of three members from each organization investigated means of merging the two associations. In 1969 this joint committee's

report suggested two forms of alliance: merger or federation. A new committee, the Joint Committee to Study Merger, developed the working paper "ASIS/SLA Proposed Implementation Plan." In fall 1970 both ASIS and SLA felt that the next step was to ascertain the feelings of the two memberships on the subject of merger. For this purpose, the Joint Committee prepared a questionnaire for the members of both organizations.

Based on the results of the questionnaire, the Special Libraries Association Executive Board voted on June 7, 1971, to discontinue merger negotiations. The American Society for Information Science Council interpreted the questionnaire results as indicating a desire for closer relationship with the Special Libraries Association but not merger. The Council established a Committee on Inter-Society Cooperation (CISCO) in summer 1971 to seek ways to expand the activities of ASIS with other organizations concerned with the transfer of information.

Due to the extremely broad range of interdisciplinary interests of its members, ASIS maintained a number of liaisons, on both a formal and informal basis, with other professional organizations through the Liaison Committee of the society. Organizations with which ASIS established liaisons included the American Association for the Advancement of Science (AAAS), the American Federation of Information Processing Systems (AFIPS), and the American Library Association (ALA).

In October 1969 ASIS presented a proposal to the United States Office of Education for operation of the Educational Resources Information Center/Clearinghouse for Library and Information Science (ERIC/CLIS). The United States Office of Education awarded the \$180,000 annual contract to ASIS commencing January 1, 1970 [24]. ERIC/CLIS was responsible for the acquisition and processing of documents on the operation of libraries and information centers, the technology used to improve their operation, and the education and training of librarians and information scientists. The society operated ERIC/CLIS by assigning specific jobs such as the production of state-of-the-art reports, specialized bibliographies, and other information services to individuals and firms. The Special Interest Groups of the society helped to identify specialists capable of handling these assignments.

A merger of the subject areas of the Clearinghouse for Library and Information Science and the Clearinghouse on Educational Media and Technology resulted in the Educational Resources Information Center/Clearinghouse on Information Resources (ERIC/CIR). The National Institute of Education awarded Stanford University the contract for the operation of the Clearinghouse on Information Resources commencing January 1, 1974. The American Society for Information Science, originally included as a subcontractor on Stanford's bid for the consolidated clearinghouse, was dropped by Stanford late in the preaward stages [25]. However, through the loss of this contract, the leadership of ASIS learned that it had never been as important to the society's membership as it was to the leadership [26].

In January 1968 CCM (Crowell Collier Macmillan) Information Sciences assumed the administration and development of the Auxiliary Publication Service under the auspices and policy direction of ASIS. Auxiliary Publication Service, the oldest activity of the society, expanded and improved under its new name, National Auxiliary Publications Service. Its primary purpose remained to enable editors to provide their readers, promptly and inexpensively, auxiliary materials that could not otherwise be obtained.

The Publications Program, created in January 1972, coordinated all ASIS publications through an office in the society's headquarters. Joshua I. Smith served as director of the Publications Program. Publications in the program include the *Journal of the American Society for Information Science* (JASIS), known as *American Documentation* until 1970. As the official journal of the society, it concentrates on reports of research and developments in information science. The *Bulletin of the American Society for Information Science* can be traced to the *ADI Newsletter*, which became the *ASIS Newsletter*. The *Bulletin* provides members with news about the people, programs, and events of the society and the information world at large.

Information Science Abstracts, formerly *Documentation Abstracts*, is a comprehensive source of information reporting literature in the field of information science and related areas. The *ASIS Handbook and Directory*, published yearly, is a directory of the society's members but also contains a short synopsis of the whole organization. Other ASIS publications include reports resulting from special symposia, workshops, and conferences organized to focus attention on specific problems, and various Chapter or Special Interest Group newsletters containing news and items of interest to their members.

Awards began to play a more important role in ASIS. Awards added in 1969 included the Best *Journal of the American Society for Information Science* Paper Award, the Outstanding Information Sciences Movie Award, the Best Information Sciences Book Award, the Best Publication by a Chapter or Special Interest Group, and the Best American Society for Information Science Student Paper Award.

The American Society for Information Science became known for its ventures in news formats and content at its annual meetings. In 1968 *American Documentation* Paper Forums provided attendees with the opportunity of meeting the authors who had published papers in *American Documentation*. A Working International Information Retrieval Network, featured in 1972, provided access to approximately 10 data bases and international access to the European Space Reorganization files. The National Science Foundation provided partial support for this project. At the 1973 meeting each registrant received personalized selective dissemination of information service to enable them to better determine which sessions to attend. The "epitome," a 500- to 700-word contributed paper summarizing unpublished work, work in planning, or work in progress, constituted the bound volume of the *Proceedings of the 1973 Annual Meeting*.

In 1972 ASIS instituted a midyear regional meeting, not limited to a geographical region and nationally sponsored. This, in essence, was a miniconference designed to complement rather than compete with the annual meeting. It dealt intensively with specific and relatively narrow topics. The midyear regional meeting became an annual function of ASIS.

The years 1968 to 1973 proved to be very productive for ASIS, both in membership and in activities. The sound financial condition of the society helped to broaden its areas of interests. ASIS developed into the most representative professional organization in the field of information science.

1974–1983

The years 1974 to 1983 brought a reduction in the programs of ASIS due to financial difficulties. During this time, the society was actively involved with the White House Conference on Library and Information Services (WHCLIS), formulation of a national information policy, and continuing education of the information scientist. ASIS continued its liaisons with other professional associations and became the national member of the International Federation for Documentation again in 1977.

After maintaining financial stability and supporting tremendous growth since the mid-1960s, ASIS was once again close to bankruptcy in 1976. Curtailment of the publication of its journals, reduction of its headquarters staff, and a request for early payment of members' dues failed to bring the society out of its financial crisis. ASIS then decided to sell its printed assets in order to avoid bankruptcy.

Two publishers, John Wiley & Sons, Inc., and Knowledge Industry Publications, Inc., will take over various ASIS publications and inventory. Wiley paid \$349,000 as an advance against royalties for the *Journal*.... This agreement runs to the year 2010.

For the next five years, KIP [Knowledge Industry Publications, Inc.] will publish the *Annual Review of Information Science and Technology* (vols. 12–16) and the *Proceedings* (vols. 14–18); it will have first refusal on new monograph manuscripts and will distribute backlist titles. The advance against royalties by KIP was \$10,000.... ASIS will retain editorial control over its *Journal*, the *Annual Review*, and the *Proceedings*, while the two publishers defray production, distribution, and other costs of the publications [27].

Hence, ASIS was diminished to an advisory role in its major program, publications.

A national information policy became a major topic in the mid-1970s. The keynote session at the 1975 ASIS annual meeting focused on the broad implications of information privacy. A 1978 American Library Association draft statement, "Toward a Conceptual Foundation for a National Information Policy," put forth the opinion that "all information must be available to all people in all formats purveyed through all communication channels and delivered at all levels of comprehension" [28]. ASIS believed the American Library Association draft statement ignored aspects of a national information policy developed previously by individual researchers and government commissions. In June 1981 the ASIS executive director testified at a United States House of Representatives hearing for the establishment of an Institute for Information Policy and Research. ASIS believed that the federal government should conduct information policy research to ultimately impact policy making on both the national and international levels.

The American Society for Information Science devoted substantial effort during the late 1970s to the White House Conference on Library and Information Services, which was held November 15–19, 1979, in Washington, D.C. Herbert B. Landau, ASIS president-elect, served as the society delegate, and Samuel Beatty, executive director of ASIS, served as the official observer to the conference. The National Commission on Libraries and In-

formation Service (NCLIS) and ASIS sponsored three meetings during 1979 and 1980 for heads of national information organizations regarding the White House Conference on Library and Information Services.

These meetings are being held to: a) discuss ways to involve each organization effectively in the White House Conference process; b) develop positions on policy with input from organizations that will be accepted by the White House Conference as formal input to its planning; and c) determine the roles of these participating organizations in implementing recommendations of the White House Conference [29].

The first meeting, held May 29-30, 1979, in Washington, D.C., was attended by representatives from 31 information-related associations including the American Library Association (ALA), the American Association for the Advancement of Science (AAAS), the American Federation of Information Processing Societies (AFIPS), and the Special Libraries Association (SLA). Eight issues, formulated by ASIS, were presented and accepted by these representatives for debate at the White House Conference on Library and Information Services:

- 1) information as a national resource; 2) the need for a national information policy; 3) the application of technology to improve library and information services; 4) the provision of information for the citizen at the local level; 5) the public-private interface; 6) the cost of access to information; 7) international activities; and 8) research for information science [30].

The second meeting in the series was held in August 1979. Each organization discussed its involvement in the White House Conference at this meeting.

The organizational representatives reviewed the resolutions passed by the White House Conference on Library and Information Services delegates and considered legislative proposals for a national library and information services program at the third meeting, in February 1980. Participants selected four priority resolutions as consensus guidelines for immediate legislative action. These were access to library and information services, technology and uniform standards, networking, and telecommunications networks. These legislative initiatives were presented in draft form for review with the White House Conference staff and the National Commission on Libraries and Information Science [31].

Copyright, an issue since the beginning of ASIS, surfaced again in the late 1970s. The society maintained a position of neutrality because of the diversity of interests and work situations of its members. However, ASIS cooperated with the Register of Copyrights, who held hearings on Section 108 of the new copyright law in conjunction with the society's 1980 annual meeting.

The Special Interest Groups of the society continued to present notable programs at the ASIS annual meetings, which also featured general sessions and exhibits. Due to the limited financial resources available during this time, the annual meetings became less innovative but remained a major function of the society. At the 1980 annual meeting a new annual award, the

Outstanding Information Science Teacher Award sponsored by ASIS and the Institute for Scientific Information, was first presented to F. Wilfred Lancaster.

Continuing education of information scientists became a focal point of ASIS. The society designed a continuing-education program exclusively for society members:

to expose those who are interested in managing the affairs of ASIS to the basic skills of leadership, administration, and management....The program encompasses a series of didactic and training modules to be offered alternatively at the ASIS Mid-Year and Annual Meetings [32].

The society began this program, offered free to its members, at the 1979 annual meeting.

Key managerial positions in ASIS changed hands during the 1970s. Samuel Beatty succeeded Joshua I. Smith in 1977, serving first as managing director, then as executive director. Martha E. Williams followed Carlos A. Cuadra as the editor of the *Annual Review of Information Science and Technology* commencing with the 1976 volume. Charles T. Meadow succeeded Arthur W. Elias as the editor of the *Journal of the American Society for Information Science* in 1977. Lois Lunin served as *Bulletin of the Society for Information Science* editor from 1974 to 1979. Anne A. Armstrong became the *Bulletin* editor in 1980.

AMERICAN SOCIETY FOR INFORMATION SCIENCE TODAY

The membership of ASIS—which in 1983 totaled over 100 institutions and 4,500 individuals including 500 students—embraces a variety of special interests in information science. The members are engaged in activities ranging from classification and coding systems to the design of information networks, from computational linguistics to cybernetics and information theory.

In 1980 ASIS conducted a survey of its members. From this, a profile of people working in the information science field can be derived. The typical society member is a 43-year-old female holding a Master of Library Science (M.L.S.) with 15 years of experience since receiving her last degree. This member, averaging an income of \$31,200, is employed in an administrative or management position in the industrial sector, with the primary function being management of information programs or services [33].

The four categories of membership in the society are: (a) regular, for which there are no formal education requirements; (b) student, in which the member must be regularly enrolled at a college or university in a program of documentation, library science, or information science; (c) retired, in which the member has been an ASIS member for the last 5 years and is retired from active work; and (d) institutional, for either profit or non-profit organizations. Both local and student chapters are located throughout major regions in the United States and Canada. "Chapters provide a means for personal contact, professional stimulation and information ex-

change on the local level by holding their own meetings on a regular basis and publishing newsletters and materials" [34]. The following Chapters (with date of chartering) now exist (as of 1983):

Central New Jersey (1975)	Northern Europe (1974)
Central Ohio (1961)	Northern Ohio (1963)
Chicago (1961)	Pacific Northwest (1971)
Colorado Centennial (1979)	Pittsburgh (1961)
Delaware Valley (1960)	Potomac Valley (1960)
East Tennessee (1975)	San Francisco Bay Area (1961)
Georgia (1980)	Southern (1976)
Indiana (1961)	Southern Ohio (1962)
Los Angeles (1961)	Taipei (1982)
Metropolitan New York (1960)	Texas (1974)
Michigan (1969)	Upstate New York (1962)
Minnesota (1968)	Western Canada (1969)
New England (1961)	Wisconsin (1974)

Student Chapters are encouraged as a means of furthering interest in the information science professions. These student chapters consist of groups of student members at various colleges and universities, with a regular member serving as adviser. Student Chapters (with year of chartering) are currently (as of 1983) established at:

Alberta, University of (1977)	Minnesota, University of (1979)
Capital Area (Washington, D.C.) (1967)	Nashville University Center (1970)
Case Western Reserve University (1966)	Northern Texas (1982)
Central New York (Syracuse) (1976)	Pittsburgh, University of (1966)
Columbia University (1964)	Pratt Institute of Technology (1970)
Denver, University of (1980)	Rutgers, University of (1964)
Drexel University (1964)	Saint John's University (Jamaica, N.Y.) (1972)
East Texas State University (1977)	San Jose State University (1982)
Hawaii, University of (1981)	Simmons College (1975)
Illinois, University of, at Urbana-Champaign (1978)	Southern California, University of (1982)
Indiana University (1975)	State University of New York at Albany (1974)
Kent State University (1975)	Tennessee, University of (1982)
Long Island University (1977)	Texas, University of, at Austin (1977)
Los Angeles Area (1976)	
Michigan, University of (1972)	

Special Interest Groups offer society members the opportunity to share their experiences with colleagues of like concern in particular disciplines of information science. The following Special Interest Groups now exist (as of 1983):

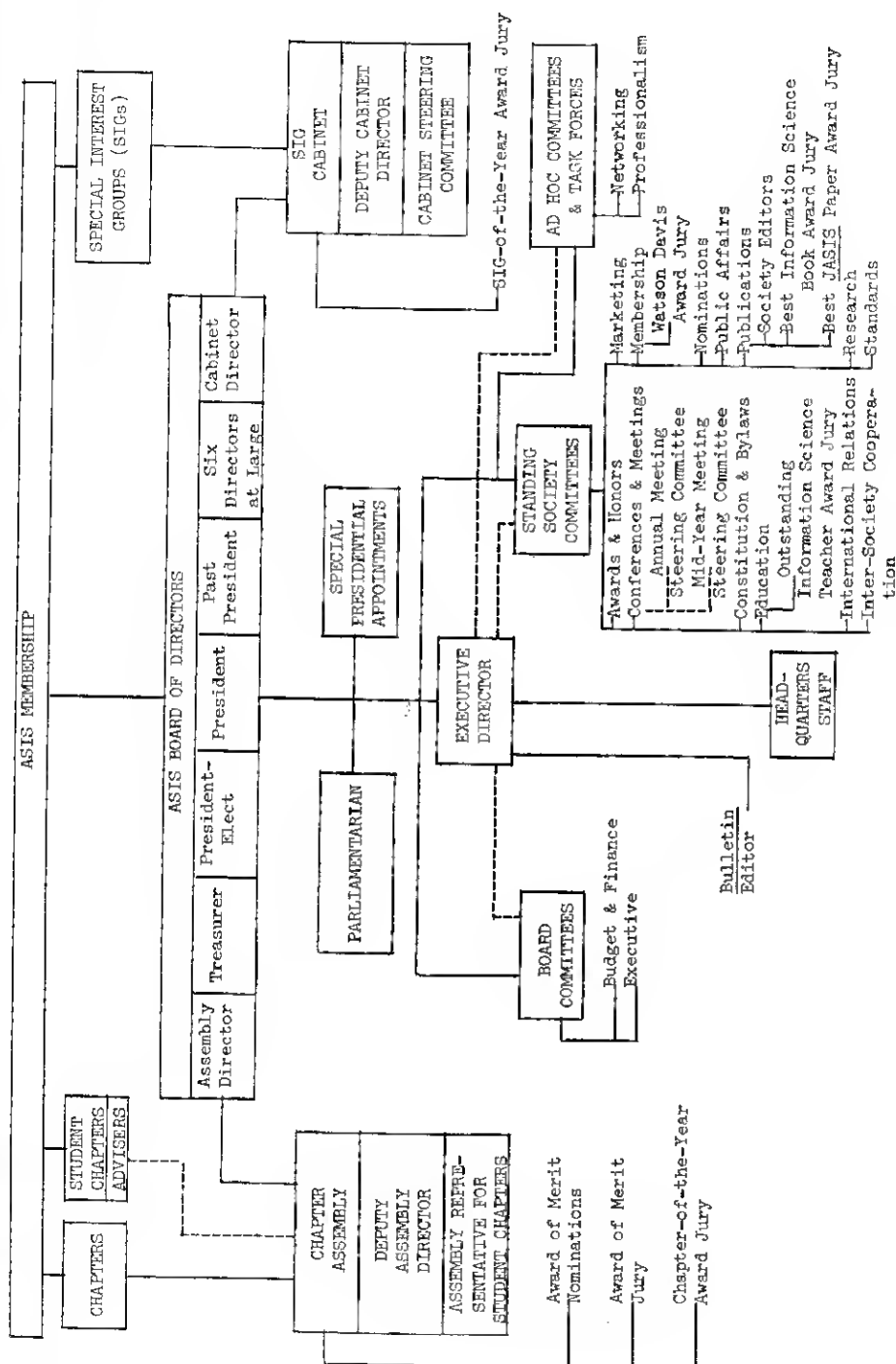


FIGURE 1 ASIS organization chart.

Arts and Humanities	Information Generation and Publishing
Automated Language Processing	Information Services to Education
Automated Office of the Future	International Information Issues
Behavioral and Social Sciences	Law and Information Technology
Biological and Chemical	Library Automation and Networks Management
Information Systems	Medical Information Systems
Classification Research	Numeric Data Bases
Computerized Retrieval Services	Public-Private Interface
Education for Information Science	Storage and Retrieval Technology
Energy and Environment	Technology, Information, and Society
Information	User On-Line Interaction
Foundations of Information Science	
Information Analysis and Evaluation	

There are three levels within the organizational structure of ASIS. The members are the lower level of the organization, the Chapters and the Special Interest Groups are the middle level, and the Board of Directors is the upper level. The Chapter Assembly provides a medium of communication between the Chapters and the Board, whereas the Special Interest Groups Cabinet is the medium between the Special Interest Groups and the Board. Thus, the members communicate with the Board directly or via the Chapters and Special Interest Groups. The Board of Directors governs the American Society for Information Science. This Board is composed of the officers, the Assembly director, the Cabinet director, and six directors-at-large. Committees of the society conduct activities dealing with the interest of ASIS in specific areas. The organization chart presented in Figure 1 depicts the structure and identifies committees of ASIS. The society maintains a national office located in Washington, D.C., with an executive director and staff.

Today the ASIS is an active and growing association dedicated to the creation, organization, dissemination, and application of knowledge relating to information and its transfer. The society acts as a bridge between research and development and the requirements of various types of information systems. ASIS provides a forum for the discussion, publication, and critical analysis of work dealing with the theory and practice of the components involved in the communication of information [35].

REFERENCES

1. *The American Society for Information Science*, ASIS, Washington, D.C., n.d., p. 1.
2. Watson Davis, *Activities of Science Service in Scientific Documentation*, 2nd ed. (Documentation Division, Document No. 72), Science Service, Washington, D.C., February 10, 1936, p. 1.
3. Claribel R. Barnett, "The Bibliofilm Service," *Agric. Lib. Notes*, 11, 55 (January 1936).
4. Claire K. Schultz and Paul L. Garwig, "History of the American Documentation Institute—A Sketch," *Am. Doc.*, 20, 154 (April 1969).
5. Barnett, Ref. 3, p. 59.

6. Davis, Ref. 2, p. 1.
7. Watson Davis, "15 Years Experience with Auxiliary Publication," *Am. Doc.*, 2, 87 (April 1951).
8. "Documentation Institute," *Am. Lib. Assoc. Bull.*, 31, 699 (October 15, 1937).
9. "Bibliofilm Service," *Lib. J.*, 63, 902 (December 1, 1938).
10. Cuthbert Lee, "American Documentation Institute and Auxiliary Publication," *Inter-American Bibliographical and Library Association Proceedings*, Vol. 2, 1939, p. 295.
11. Cuthbert Lee, "Le Bibliofilm de la Commission Internationale des Industries Agricoles," *Am. Lib. Notes*, 14, 359 (June 1939).
12. Henry J. Dubester, "The Role of the American Documentation Institute in International Documentation," *Am. Doc.*, 3, 117 (January 1952).
13. "Bibliofilm Service," *Agric. Lib. Notes*, 16, 429 (August 1941).
14. "Minutes of the Deferred Annual Meeting of ADI," *Am. Doc.*, 3, 7 (January 1952).
15. G. Miles Conrad, "Minutes of the Council Meetings," *Am. Doc.*, 3, 205 (October 1952).
16. C. Dake Gull, "Annual Report of the President of the American Documentation Institute," *Am. Doc.*, 12, 4 (January 1961).
17. James E. Bryan, "ADI Today and Tomorrow," *ADI Newsl.*, 5, 1 (September–October 1966).
18. John B. Kaiser, "A New 'Newsletter'," *ADI Newsl.*, 1, 2 (May 1961).
19. "1965 ADI Annual Business Meeting," *ADI Newsl.*, 4, 2–3 (November–December 1965).
20. American Society for Information Science, 1969–70 *American Society for Information Science Handbook and Directory*, ASIS, Washington, D.C., 1969, p. ii.
21. James E. Bryan, "'By Another Name...'," *ADI Newsl.*, 6, 2 (July–August 1967).
22. "Amended ASIS Constitution," *ASIS Newsl.*, 7, 2 (November–December 1968).
23. "SLA/ASIS Merger Discussions," *Special Lib.*, 61, 387 (September 1970).
24. "ASIS Takes Over ERIC/CLIS and Names New Director," *Lib. J.*, 95, 614 (February 15, 1970).
25. "ASIS Unhappy with ERIC Contract Decision," *ASIS Newsl.*, 19, 6 (March 1974).
26. Herb White, "Message from Herb White," *ASIS Newsl.*, 12, 2 (November 1973).
27. "ASIS: 'Basic Black with No Frills'," *Wilson Lib. Bull.*, 52, 17 (September 1977).
28. Audrey N. Grosch, "ASIS Marching Forward: Public Policy and the Profession," *Bull. Am. Soc. Inf. Sci.*, 4, 8 (February 1978).
29. "Information Community Organization Heads Pre-White House Conference," *Bull. Am. Soc. Inf. Sci.*, 5, 30 (August 1979).
30. Ref. 29, p. 31.
31. "Implementing White House Conference Resolutions," *Bull. Am. Soc. Inf. Sci.*, 6, 31 (April 1980).
32. James M. Crestos, "Interested in Managing? Try ASIS," *Bull. Am. Soc. Inf. Sci.*, 5, 4 (February 1979).
33. Donald W. King, Cheri Krauser, and Virginia M. Sague, "Profile of

- ASIS Membership," *Bull. Am. Soc. Inf. Sci.*, 5, 9-17 (August 1980).
34. *The American Society for Information Science*, Ref. 1, p. 3.
35. *Encyclopedia of Associations*, 7th ed., s.v., "American Society for Information Science (ASIS)."

APPENDIX 1: PRESIDENTS OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE

1937-1943	Watson Davis	1954	Milton O. Lee
1944	Keyes D. Metcalf	1955	Scott Adams
1945	Waldo G. Leland	1956	Joseph Hilsenrath
1946	Watson Davis	1957	James W. Perry
1947	Waldo G. Leland	1958	Herman H. Henkle
1948-1949	Vernon D. Tate	1959	Karl F. Heumann
1950-1952	Luther H. Evans	1960	Cloyd Dake Gull
1953	Eugene Miller	1961	Gerald J. Sophar
1962	Claire K. Schultz	1974	Herbert S. White
1963	Robert M. Hayes	1975	Dale B. Baker
1964	Hans Peter Luhn	1976	Melvin S. Day
1964-1965	Laurence B. Heilprin	1977	Margaret T. Fischer
1966	Harold Borko	1978	Audrey N. Grosch
1967	Bernard M. Fry	1979	James M. Crestos
1968	Robert S. Taylor	1980	Herbert B. Landau
1969	Joseph Becker	1981	Mary C. Berger
1970	Charles P. Bourne	1982	Ruth L. Tighe
1971	Pauline A. Atherton	1982-1983	Charles H. Davis
1972	Robert J. Kyle	1984	Donald W. King
1973	John Sherrod		

APPENDIX 2: HISTORICAL DATA ON THE AMERICAN DOCUMENTATION INSTITUTE/AMERICAN SOCIETY FOR INFORMATION SCIENCE

Significant Events

1934	Bibliofilm Service established
1936	Auxiliary Publication Service established
1937	Founding of the American Documentation Institute
1946	<i>Catalog of Auxiliary Publications in Microfilms and Photoprints</i> , first edition, published
1950	Publication program established: first issue of <i>American Documentation</i>
1952	Individual membership granted into ADI
1954	Annual meeting held away from Washington, D.C., for the first time
1954	First official ADI Award presented to Dr. Atherton Seidell
1960	ADI Secretariat established
1960	First Chapter, Potomac Valley Chapter, chartered

(continued)

APPENDIX 2 (Continued)

-
- | | |
|-----------|---|
| 1961 | ADI logo adopted |
| 1961 | First Newsletter issued |
| 1962 | Placement Service established |
| 1964 | First <i>Proceedings</i> issued: <i>Proceedings of the 26th Annual Meeting</i> , first set of technical articles ever automatically photocomposed |
| 1964 | First Student Chapters chartered: Columbia University, Drexel University, and Rutgers University |
| 1964 | Scholarship Fund for Information Science established in memory of Hans Peter Luhn |
| 1964 | First Award of Merit presented to Hans Peter Luhn (posthumously) |
| 1965 | Sponsored the Working Symposium on Education for Information Science (Airlie Conference) |
| 1966 | First <i>Annual Review of Information Science and Technology</i> published |
| 1966 | First issue of <i>Documentation Abstracts</i> published |
| 1966 | First Special Interest Groups established: |
| | Automated Language Processing |
| | Behavioral and Social Sciences |
| | Biological and Chemical Information Systems |
| | Classification Research |
| | Education for Information Science |
| | Information Analysis Centers |
| | Library Automation and Networks |
| | Reprographic Technology |
| 1968 | American Documentation Institute became American Society for Information Science |
| 1970–1973 | Operated Education Resources Information Center/Clearinghouse for Library and Information Science |
| 1971 | ASIS logo adopted |
| 1979 | Participated in the White House Conference on Library and Information Services |

ADI/ASIS addresses in Washington, D.C.

- | | |
|-----------|---|
| 1937 | Science Service, National Academy of Sciences |
| 1950s | Library of Congress |
| 1960–1961 | 1025 Connecticut Avenue, N. W. |
| 1962–1963 | 1728 N Street, N.W. |
| 1964–1968 | 2000 P Street, N.W. |
| 1968–1970 | 2011 I Street, N.W. |
| 1970–1973 | 1140 Connecticut Avenue, N.W. |
| 1974–1978 | 1155 16th Street, N.W. |
| 1978– | 1010 16th Street, N.W. |

Continuing Publications

American Documentation (1950–1968); *Journal of the American Society for Information Science* (1969–)

(continued)

APPENDIX 2 (Continued)

ADI Newsletter (1961–1967); *ASIS Newsletter* (1968–1974); *Bulletin of the American Society for Information Science* (1974–)
Proceedings of the Annual Meeting (1964–)
Annual Review of Information Science and Technology (1966–)
Documentation Abstracts (1966–1968); *Information Science Abstracts* (1969–)

Executive Directors

1960–1963	John B. Kaiser	1970–1973	Herbert Koller
1963	Paul Janaske	1973–1977	Joshua I. Smith
1964–1969	James Bryan	1977–	Samuel Beatty

ANGELA DIERKING REDMOND

AMETEK: HOUSTON INSTRUMENT DIVISION

INTRODUCTION

Throughout its 27-year history, computer peripherals manufacturer Houston Instrument has tracked the development of computers and microprocessors through its state-of-the-art products—*analog recorders and intelligent plotters and digitizers.*

FOUNDING MANAGEMENT

Founded in 1959 by Edward V. Hardway, Jr., in Houston, Texas, the company had three employees, five sales representatives, and one product—an *analog recorder.*

Today, Houston Instrument, under the direction of President L. C. Bower, who joined the company in 1966 as vice president of sales, has nearly 500 employees, a worldwide sales and service support network, and over 30 major products in three distinct product lines. Now headquartered in Austin, Texas, the market pacesetter for both test and analytical instruments and computer peripherals maintains manufacturing facilities in both Austin and Gistel, Belgium.

EARLY PRODUCT DEVELOPMENT

Houston Instrument made its first mark producing an analog recorder—the HR-92 Omnigraphic X-Y recorder—designed for laboratory and test applications.

Analog recorders are electromechanical instruments designed to record two variables, such as temperature versus pressure, using ink on paper. Basically used to plot data in Cartesian coordinate form on graph paper, analog recorders are powerful and productive measurement tools found in almost every field of science and industry.

As the company grew, it diversified its product lines, adding digital plotters, graphics digitizers, and strip-chart recorders.

SIGNIFICANT COMPANY EVENTS

There were several key milestones that fueled Houston Instrument's upward mobility in establishing itself as a forerunner in the microcomputer peripherals market. These significant events include its purchase by Bausch & Lomb in 1967 and by Ametek in 1985, and the firm's relocation to what has become the high-tech boom town of Austin.

Based in Rochester, New York, Bausch & Lomb, a giant in the eye care industry, purchased Houston Instrument in 1967 and decided to establish the rapidly growing manufacturing firm as a graphics division.

Already a leader in the analog recorder industry at the time, Houston Instrument had made its first entry into the computer peripherals market in 1965 with the introduction of the COMLOT line of sophisticated digital plotters. These plotters were specifically designed for a wide range of design applications.

While continuing to track the growth of the instrumentation market, as well as the microprocessor and computer industries, Houston Instrument introduced the OmniScribe strip-chart recorder. Used as a readout device, such as a panel meter, for analytical instruments, the strip-chart recorder measures one or two variables versus time.

As the company expanded its analog product lines and continued its push into the pen plotter market, it soon outgrew its facilities in Houston and moved to Austin in 1974.

Shortly after relocating its headquarters to Austin, Houston Instrument announced its electrostatic products, which included line printers, plotters, and CRT hard copy devices. Instead of the standard method of putting line traces on paper with pen and ink, these products used an "electrostatic" principle.

With this method of plotting, an electrostatic charge is placed on the paper through a row of styli that are energized where marks on the paper are desired. The charged paper is then pulled over a toner supply bar by drive rollers. The charged area picks up toner particles from the bar, making the writing visible.

To meet the new, more sophisticated recording requirements in the laboratory and process control markets, Houston Instrument introduced another method of recording—the thermal plotter/printer—using thermal paper to record graphs and print characters.

The thermal plotter/printer represented a compromise between the analog strip-chart recorder and the digital plotter and made it possible to print data in a normal fashion as well as annotate a curve much as the operator would do with a pencil. The first applications of these thermal writing devices were in gas and liquid chromatographic equipment.

The company took advantage of the potentially explosive personal computer market in 1979 with the introduction of its innovative line of low-cost, high-quality microcomputer-compatible plotters and digitizers.

In late 1983, in an effort to concentrate on the health care arena, parent company Bausch & Lomb decided to divest its Instruments Division, of which Houston Instrument was one component. The computer peripherals manufacturer was purchased nearly 2 years later by Ametek, Inc.

Ametek is a Fortune 500 diversified manufacturer with 36 manufacturing locations nationwide. The company produces both capital goods and components used in consumer products through its four operating divisions: electro-mechanical, precision instruments, process equipment, and industrial materials.

PRODUCT DEVELOPMENT TOWARD MICROCOMPUTER MARKET

As the personal computer evolved into a multibillion dollar industry, a cadre of add-ons—input/output devices—designed to enhance system capabilities

also entered the consumer marketplace. Although sales to industry continued to be important, Houston Instrument began concentrating on the booming personal computer industry, expanding its product lines to meet the demands of established and developing markets.

OVERVIEW OF PEN PLOTTERS AND GRAPHICS DIGITIZERS

The pen plotter is the most common method of recording digital data. Based on mechanical designs pioneered in the development of analog X-Y recorders, this recording device takes data from a computer or processor and translates it into actuating signals, which cause incremental movements of the plotter pen to convert the data into a drawing. This graphic portrayal of digital data is produced by a combination of X and Y carriage movements.

Pen plotters have dominated the market as the most efficient means of producing hard copy graphics. The advantage pen plotters have over other more recent developments in graphics-production equipment is that pen plotters offer the user greater flexibility at a lower cost.

Graphics digitizers, long seen as viable data-entry devices in architectural and engineering professions, can also serve as useful alternatives to the keyboard in the medical, mapping, environmental, geologic, and industrial fields.

In a sense, a digitizer can be considered the electronic version of a sheet of grid paper. The digitizer first determines the X-Y coordinates of a point electronically and then transmits this coordinate data in digital form to the host computer or processor.

Digitizers are among the most powerful peripherals on the market today because they enable the user to control the cursor and enter computer commands or graphics data without having to resort to the keyboard. In many cases, these input devices literally have replaced the keyboard as the primary means of entering data into a computer, particularly in computer-aided design (CAD) configurations.

Precision, accuracy, and flexibility have made the digitizer the preferred choice over other input devices such as the mouse, the light pen, the joystick, the trackball, and the touch screen.

TARGET MARKETS

Houston Instrument's highest growth area is its line of micrographic plotters and digitizers. As a strong vertical marketer, Houston Instrument also concentrates heavily on the PC-based CAD arena with peripherals that complement a wide range of drafting and design-intensive applications.

The company's family of intelligent digital plotters and line of graphics digitizers are designed to meet the needs of professionals in business, education, architecture, engineering, and a host of other drafting-intensive applications, such as facilities design and management and technical illustration.

Houston Instrument's success in the highly competitive PC CAD market was partly due to the extensive compatibility its products have with third-party software programs. Through its "HI Connection" program, the company works with software vendors to ensure that its products are compatible with

a variety of graphics software packages. As a result, Houston Instrument products are immediately compatible with more than 350 commercially available CAD and business graphics software packages.

CURRENT PRODUCT LINES

The company's compact and versatile four-pen PC Plotter Series and the single-pen DMP-40 and eight-pen DMP-29 plotters are specifically designed for the low-priced market, which is focused primarily on microcomputer users—mainly business people—who want to make multicolor overhead transparencies or produce hard copy graphs and charts for business presentations.

Houston Instrument's family of large-format CAD plotters—the A- to E-size DMP-56A, the C- and D-size single-pen DMP-41/42 and DMP-51/52 and 14-pen DMP-51/52 MP—are designed for architects and engineers who need to produce detailed architectural floor plans, mechanical designs, or electrical schematics.

Entering a new area in late 1986, the company introduced the first wide-format automatic scanner for under \$3,000. Called SCAN-CAD Plotter Accessory Model 128, the unit is an accessory to Houston Instrument's DMP-50 Series plotters and streamlines entering graphics data into computers.

The company also manufactures a complete line of graphics digitizers for use in a wide range of applications from cursor control and menu selection to mechanical, electrical, surveying, mapping, and architectural applications. These graphics input tools include the low-cost, high-performance TRUE GRID 1000 and 8000 Series and the HIPAD Series.

The TRUE GRID digitizers offer users the ability to choose the appropriate resolution, accuracy, and transducer to suit a specific application. Based on electromagnetic technology, these tablets are designed for menu selection or for complementing a CAD configuration.

Like the TRUE GRID tablets, the HIPAD Series also enables users to quickly digitize and modify intricate drawings and translate these graphics into computer data.

Established as a manufacturer of analog recorders nearly three decades ago, Houston Instrument has also continued to be a market pacesetter in the test and instrumentation industry. Its analog product lines—the Series 100, 200, and 2000 X-Y recorders, and the Series 4500 MicroScribe and Omni-Scribe D-5000 strip-chart recorders—command a healthy market share.

FUTURE TRENDS OF COMPUTER

PERIPHERALS INDUSTRY

A report by International Data Corporation of Framingham, Massachusetts, indicates that the printer and plotter market is expected to grow 26.4% by 1990. Trends that will impact this anticipated growth include changing graphics needs, an increased demand for color hard copy, and improved output technologies.

In 1984, pen plotters represented 4.9% of a \$3 billion industry, according to a report by industry analyst Future Computing. The same report estimates that by 1990, pen plotters will command 9.4% of an almost \$10 billion industry.

The forecast for graphics digitizers also looks optimistic. A report by Predicasts, Inc., estimates that digitizers will account for \$230 million in sales in 1987 and \$490 million in sales by 1995. According to Predicasts, digitizer sales will grow at an annual rate of nearly 27% through 1987.

MARTY McCONNELL

ANALOG-TO-DIGITAL AND DIGITAL-TO-ANALOG CONVERSION

The measurements and signals that occur in nature or in man-made devices are analog or digital in form. Analog quantities can take on any values within a given range. Most physical measures (i.e., temperature, length, voltage) are analog. Digital quantities may take only specific values within a range. The binary signals that computers and telecommunications systems use are digital signals. Devices that are able to take analog signals and convert them to digital signals are called analog-to-digital (A/D) converters. Likewise, digital-to-analog (D/A) converters convert digital signals to analog signals.

A/D and D/A converters have many applications and are found in a wide variety of common devices, such as radios, instruments, and appliances. Because most of our computational devices are digital, perhaps the most important use for A/D and D/A converters is to allow digital computers to interact with an analog world. In a microcomputer, for example, an A/D converter is used to convert the analog position of a joystick to digital coordinate values. Computerized control systems and laboratory instrumentation systems are other significant industrial uses for A/D and D/A converters.

Figure 1 is a schematic of how a computer with A/D and D/A converters might be used to control a motor. Motor status information, speed, for example, is read during a very short time interval from the motor. This instantaneous reading of data is called a sample. These data are converted into digital form via an A/D converter and input to the control computer. The computer then makes decisions as to whether or not the motor is operating within the desired speed limits and sends digital control information, such as to speed up or slow down, to a D/A converter so that the appropriate analog control voltage can be set. We could substitute any analog device for the motor, and the diagram would serve as a representation for a general control system.

Hybrid computers are another significant use for A/D and D/A converters. A hybrid computer is composed of a digital computer and an analog computer, which are connected by A/D and D/A converters. Some types of mathematical operations (i.e., those involving systems of differential equations) can be done more effectively by analog computers. In a hybrid computer system, computational tasks would be evaluated to determine whether they should be assigned to the analog or digital section for processing. The analog and digital sections would then exchange data through A/D and D/A converters to complete an entire job.

Although the most common A/D converters convert analog electrical signals to digital electrical signals, converters are available that convert mechanical, thermal, and optical analog signals into digital electrical signals. Consider the position (or length) converter shown in Figure 2. It is composed

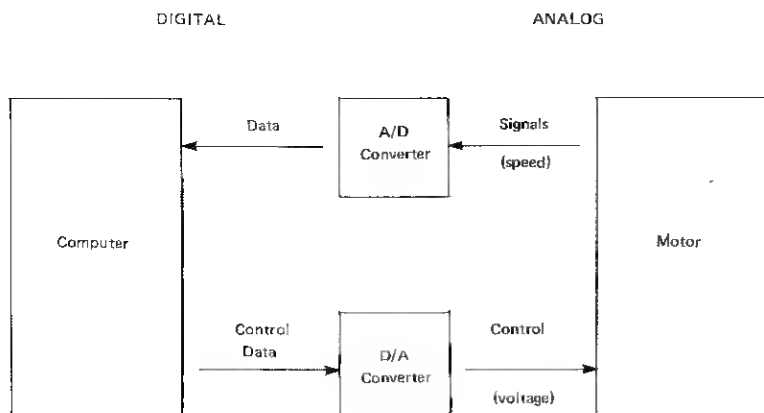


FIGURE 1 A Control system.

of a fixed component, the track, and a movable component, the slide. The track has three lanes with strips of conducting and nonconducting materials. Each conducting strip is connected to a voltage source. The slide has a contact for each lane. As the slide is moved over the track, the contacts touch either conducting or nonconducting strips. Thus, a 3-bit digital pattern of voltages appears at the terminals of the slide that corresponds to the position of the slide on the track. As one can see, if more lanes are added to this device, the digital signal corresponds more closely to the exact analog position of the slide.

D/A converters are generally simpler and easier to construct than A/D converters. Figure 3 shows a simple D/A converter. It consists of a network of resistors connected to a set of voltages controlled by digital inputs. The input digital signal D0-D5 is stored in a memory location called a register. The register is connected to switching circuitry, which switch resistance network input terminals to digital voltage sources that correspond to binary

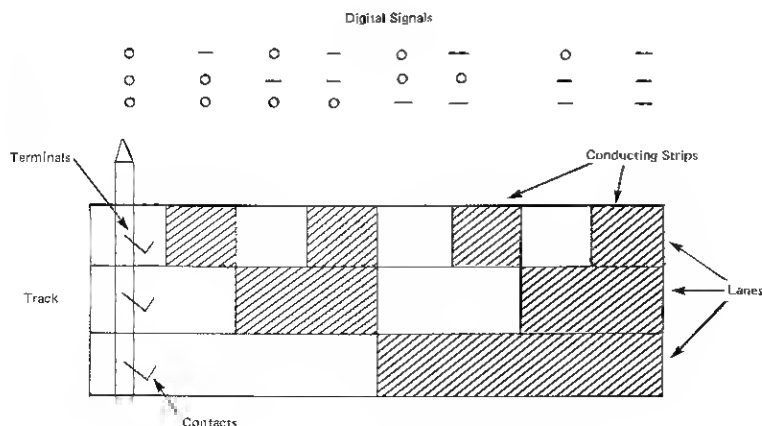


FIGURE 2 A Position A/D converter.

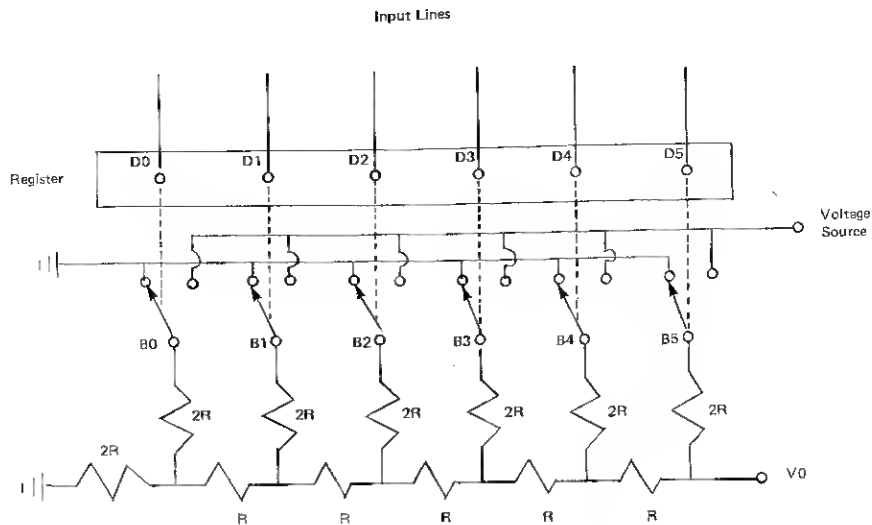


FIGURE 3 A Simple D/A converter.

data values. B0 corresponds to the least significant bit and B5, to the most significant bit. It can be shown through electrical circuit calculations that the voltage produced at the output (VO) is twice as large when B1 alone is connected to an input voltage source as when B0 alone is connected to that same source. VO is four times larger when B2 is connected than when B0 is connected, and so on. The VO doubles with each increasing bit position. It can also be shown that the VO that results from two or more of the inputs being connected is the sum of the voltages of the individual connections. By adjusting the circuit parameters, a digital signal pattern can be converted into one of $2^{**6} = 64$ analog voltages within a given range.

Many A/D converters use D/A converters as components. In these devices, analog signals are generated by feeding varying digital signals to a D/A converter. The output of the D/A converter is compared with the sampled analog input. When these two differ within an acceptable range, the generated digital signal input to the D/A converter is taken to be the correct digitization of the sampled analog signal.

A/D converters may be either serial or parallel in their conversion techniques. In a parallel A/D converter, all of the digital data bits are generated at the same time. Serial A/D converters produce a single data bit at a time starting from the most significant bit. In general, parallel converters are preferred because they are faster.

Two important factors in the selection of A/D and D/A converters are speed and resolution. Resolution is expressed as the number of bits of the device's digital signal. A higher resolution device is capable of representing more states within the analog signal range. Hence, the digital representation more closely approximates the analog signal. Integrated circuits are commercially available with resolutions of 12 bits and higher. Speed is essential where we wish to take data samples at a fast rate. It is also important when the analog signal is changing rapidly. A sampled analog signal may change significantly in value in the time it takes to convert the signal to digital form, thus giving a false reading.

BIBLIOGRAPHY

- Carr, Joseph, J., "Digital Interfacing with an Analog World," Tab Books, Blue Ridge Summit, PA, 1978.
- Cassell, Douglas A., "Microcomputers and Modern Control Engineering," Reston Publishing, Reston, VA, 1983.
- Dingwall, Andrew G. F., and V. Zazzu, "High Speed D/A and A/D Conversion," in *Proceedings of the 1984 IEEE Symposium on Circuits and Systems*, Vol. 1., Montreal, Canada, May 7-10, 1984.
- Holland, Alexander, "High Resolution, High Linearity Interpolating A/D Converter, in *Proceedings of the 1984 IEEE International Test Conference*, Philadelphia, PA, October 16-18, 1984.
- Sargent, Murray, and Richard L., Shoemaker, "The IBM Personal Computer from the Inside Out," Addison-Wesley, Reading, MA, 1984.
- Sloan, M. E., "Computer Hardware and Organization: An Introduction," Science Research Associates, Chicago, IL, 1983.

KENNETH SOCHATS

ANALYSIS OF THE ECONOMICS OF GAS AND OIL (EGO)

FOREWORD

EGO* (which stands for economics of gas and oil) is a computerized economic analysis model designed to aid oil and gas producers in assessing the financial impact of producing out wells or prospecting for new wells. EGO is an IBM PC/XT/AT-based model that is fast, reliable, and extremely flexible. The model is primarily designed to project future oil and gas production from one well or multiple wells up to the limit of the storage capacity of the hard disk. Any arbitrary combination of wells can be included in a run. The selection of the wells can be by property, cost center, or any region defined by the user. Each well can have different production rates, ownerships, oil and gas prices, operating costs, and other economic factors. EGO determines future cash flow (both before and after income tax) and then combines the results, for any or all the wells, on a printout that even bankers and financial analysts will sanction. Future enhancements of the model will also provide the capability to set up balance sheets, income statements, and other relevant financial statements.

INTRODUCTION

Every company has the goals of making a healthy profit (and thus increasing wealth) and continuing to generate new business in order to grow. For an oil- and gas-producing company, this means, first, producing out current wells as long as it is profitable to do so and, second, finding and acquiring new reserves. This article will discuss the basic concepts required in evaluating reserves and will describe the microcomputer-based model called EGO.

The next section will describe fundamentals of before tax cash flow and after tax cash flow, windfall profit tax, severance and ad valorem tax, the concepts of present worth, the rate of return, and so forth, as applied to oil and gas investments. The analysis of incorporation of risk through the use of decision analysis, although an integral part of any such discussion, will not be discussed because it only complicates the understanding of the basic concept of oil and gas economics. Furthermore, because any analysis involving risk (or uncertainty) requires a thorough understanding of the case involving certainty, we feel that the interested reader can always read other material on risk (including the EGO manual). EGO does incorporate risk by including probabilities of success in drilling wells. Thus, EGO assumes that the user has been able to evaluate the uncertainty in a drilling venture and has been able to reduce to a simple probability the success of drilling.

*EGO is a trademark of Stanford Business Software, Inc., Mountain View, California.

The last section will outline the many features available through EGO, which provide the tremendous advantage that can be gained through the use of a computerized system like EGO.

BACKGROUND ON ECONOMIC ANALYSIS

The basis for all investment decisions is the ultimate or cumulative cash flow. The cash flow needs to be examined from the point of view of investments, income, expenses, before tax cash flow, and after tax cash flow. Typically, a company will look at the cash flow on a tax basis and a book basis. Depending on the use of the analysis, a company may look at the cash flow by using constant costs and prices or escalated costs and prices.

In this article we will concentrate primarily on the cash flow as reported to the IRS.

Cash Flow Basics

Because the primary concern of a company is to make a profit, we need to examine the profit of the company on a year-by-year basis by looking at the basic equation

$$\text{Profit} = \text{Revenues} - \text{Expenses}$$

where revenues are the gross receipts of the company and expenses are the costs incurred to operate the company. The profit obtained can be evaluated before tax, after tax, and on the books. The main differences between tax and book reporting arise because of the manner in which expenses are treated.

The equations for analyzing cash flow on a before and after tax cash basis can be stated as follows:

$$\text{Before Tax Cash Flow} = \text{Gross Revenues to the Company}$$

$$- \text{Direct Costs to the Company}$$

$$- \text{Investments}$$

$$\text{Income Tax} = \text{Income Tax Rate} * \text{Taxable Income}$$

$$\text{After Tax Cash Flow} = \text{Before Tax Cash Flow}$$

$$- \text{Income Tax}$$

If the cash flow items are to be reported on a book basis, an item called book DD & A (which stands for depletion, depreciation, and amortization), rather than investments, is subtracted when computing the before tax cash flow.

Production

For an oil and gas company, revenues are generated through the production of gas and oil. The yearly production depends on the wells that the company owns in that year and the amount of flow that can be obtained from the wells.

Because it is impossible to predict exactly how much each well flows every year, and the production declines from the day that the well is drilled, decline curves are used to model future production. Typically, there are three commonly used decline curves in the industry: exponential, hyperbolic, and harmonic. Both the hyperbolic and exponential are commonly used in the industry, with the use of harmonic declines being not quite so common.

The curves are obtained by plotting daily production against time on semilog paper. If the curve is a straight line, the exponential decline curve represents the production accurately. On the other hand, if the straight line curves up, the curve is a hyperbolic decline curve. A special case of hyperbolic declines is harmonic declines.

The production specified by an exponential decline is as follows:

$$q_t = q_0 e^{-Dt}$$

where q_t is the production at time t , q_0 is the production time 0, D is the nominal decline rate, and e is the base of the natural logarithm and is equal to 2.71828.

Because we would generally not expect production to be accurately represented by just one curve, EGO allows the user to enter many different curves (which could be mixed between exponential, hyperbolic, and harmonic) for different portions of the production life of a well. Thus, production is actually represented through a piecewise exponential (or hyperbolic or harmonic) curve or a combination of different piecewise curves.

Revenues

Once the production for each year is known, the revenues can be computed by multiplying the price times the production (these revenues are called gross, or 8/8, revenues). However, typically, the total revenue obtained from a well is divided between a number of different parties based on the interest owned by the various parties.

Typically, an oil and gas producer will buy rights to produce oil and gas from wells on property owned by the "landowner." The land will be leased with various clauses on the lease and the time frame during which the lease is operable. Furthermore, the landowner will often require a royalty interest, which is typically 12.5%. In the simple case when only one oil and gas producer is involved, that producer will typically own 100% working interest (i.e., bear 100% of all costs) and 87.5% net revenue interest (i.e., keep 87.5% of the 8/8 revenues).

In more complicated deals, the working interest and net revenue interest are further divided between partners. Furthermore, the deal may be set up so that a particular party may get a different portion of the revenues and may pay a different portion of the cost, depending when the oil and gas is sold. The points at which the working interest and/or net revenue interest change are called reversion points. These reversion points are usually specified as points when a certain percentage (usually greater than 100%) of costs are recovered. The points in time when a reversion in interest occurs are also called payout points (not to be confused with years to payback, which is the number of years it takes to recover all expenditures).

EGO is flexible in specifying the reversion points and working/net revenue interests.

Costs and Economic Limit

Costs can be broken into two categories: capital expenditures costs and operating costs. These items are described below in a broad sense.

Capital expenditures are the investments that will generate benefits over many years. Examples of such expenditures are drilling and development costs, surface equipment costs (typically classified under drilling costs), and leasehold costs (costs to obtain a lease on the land on which the wells will be drilled).

Operating costs, on the other hand, are costs that will only benefit the period in which they are made. These costs can be further divided into fixed costs and variable costs. The fixed operating costs are items such as salary (clearly these are not entirely fixed because the salary depends on the number of staff which, in turn, depends on the number of wells owned by the company). Often the fixed costs are classified under administrative and general costs. Variable operating costs on the other hand are dependent on the actual production. However, caution must be exercised here because the common belief of allocating operating costs on a per unit (barrel or million cubic feet) basis may not be adequate as there are many situations where the production falls but the variable cost rises (e.g., the staff required to operate the well stays the same, but extra raw materials are required to pump out the remaining reserves). Thus, it is often best to use a variable cost that is on a per month basis.

So far, we have not considered how long we should operate wells. Clearly, if the cost that would be saved if the well were abandoned is larger than the revenues obtained from the well, the well should be plugged and abandoned. Thus, the decision on how long the well should operate requires comparing incremental revenues against incremental costs (variable costs plus the salaries of the staff that will no longer be used when the well is abandoned). Once the incremental revenues fall below the incremental costs, the well should be plugged and abandoned.

EGO allows the consideration fixed, variable, and operating costs on a per month basis, as well as on a per unit of production basis. In some situations, because of the particular deal, an oil/gas producer may want to produce out a well regardless of the economic life of the well; thus, EGO also provides the user with capability of not comparing the economic life or pre-specifying the economic life as either a number or the economic life plus a number.

Severance Taxes and Ad Valorem Taxes

Severance and ad valorem taxes are state and local taxes and are usually referred to as production taxes. These taxes have to be paid regardless of whether any profit is made or not. However, they are deductible from the revenues for the purpose of calculating income taxes. In some states these may be called by other names and there may be other breakdown of the taxes (e.g., conservation tax, school tax, etc.).

In EGO, the user can enter severance tax rates and ad valorem tax rates. The reason for not entering only one item called production tax is that in certain instances, a portion of the production taxes (the severance tax portion) may not be deductible for the purpose of calculating windfall profit taxes.

Windfall Profit Taxes

All domestic crude oil prices were decontrolled effective January 28, 1981. As part of this decontrol package, the Crude Oil Windfall Profit Tax was passed effective March 1, 1980. The tax is to be phased out by December 31, 1987 or

the date the revenues collected under the Act have totaled \$227.3 billion, whichever is later, but in no case later than January 1, 1991.

The windfall profit tax for crude oil works as follows. The tax varies depending on which of the "three tiers" the production is classified under and what the status of the producer is (i.e., independent or major). For each tier there is an adjusted base price. The calculation on a dollar per barrel basis proceeds as follows:

$$\begin{aligned}\text{Windfall Profit Tax} &= \text{Windfall Profit Tax Rate} \\ &\quad * (1 - \text{Severance Tax Rate}) \\ &\quad * (\text{Sales Price} - \text{Adjusted Base Price})\end{aligned}$$

Note that in some instances the severance tax adjustment may not apply or only a portion of the severance tax adjustment may apply, in which case the severance tax adjustment factor "(1 - Severance Tax Rate)" is suitably modified.

Furthermore, the "net income limitation" (NIL) rule also applies, that is, the windfall profit tax is limited to 90% of the "net income" on a per barrel basis. The NIL rule is too complicated to describe here, and the interested reader should refer to one of the references at the end of the article.

EGO has the capability of handling windfall profit tax treatment as specified under the current laws. Furthermore, EGO also allows the user flexibility in overriding some of the criteria used in computing windfall taxes.

DD & A

DD & A is standard terminology in oil/gas accounting and is usually computed differently for tax purposes and book purposes. The DD & A will be described for tax purposes, and mention will be made of what can be done for the books.

Only the successful drilling costs are considered for depreciation. The dry hole costs are always expensed and thus are not a part of the DD & A.

Tangible drilling successful well costs (e.g., wellhead production tanks) are depreciated. Under the new tax law, these costs are depreciated through a 7-year double declining balance method. However, EGO has the capability of allowing the user to specify any depreciation schedule over any time period. Thus, the effect of further tax law changes can be incorporated easily into EGO by the user.

The intangible drilling successful well costs (e.g., wages, fuel, repairs, etc.) can be either capitalized and depleted or the costs can be expensed. If they are not depleted, producers classified as major producers must amortize (depreciate) 30% of the intangible drilling costs in a straight line (using equal percentages for each month) over 5 years. The remaining 80% of the intangible drilling costs are expensed by the major producers. If producers classified as independents decide not to deplete the intangible successful drilling costs, they must expense the entire amount.

The depletable costs are depleted on a unit of production basis, meaning the amount of production in a given year, as a fraction of the total reserves remaining at the start of the year, is used to determine the portion of the depletable costs that are to be subtracted from the income for tax purposes. The costs that can be depleted are leasehold costs, delay rental costs (payments made by the lessee to the lessor to extend the terms of the lease by an

additional year to prevent the lease from running out), G & G (geological and geophysical) costs, and intangible drilling costs. Of these costs, delay rentals need not be depleted; they can be expensed at the election of the taxpayer. The intangible drilling costs also need not be depleted and a portion could be expensed and a portion amortized as described above. The calculation of the cost depleted in any year is given by the following formula:

$$\text{Depletion}[t] = \frac{(\text{Cost Remaining}[t] + \text{New Cost}[t]) \times \text{Production}[t]}{(\text{Reserves Remaining}[t] + \text{Production}[t])};$$

where Depletion[t] is the amount of cost depleted in year t; Cost Remaining[t] is the amount of cost remaining to be depleted as of the start of year t; Production[t] is total production in year t; Reserves Remaining[t] are reserves remaining as of the end of year t (this includes any new discoveries made in year t).

The method described above is called cost depletion. Another form of depletion called percentage depletion can also be used but will not be described here. For a description of this method, refer to one of the references.

For book purposes there are two methods: successful efforts accounting and full cost accounting. Successful efforts accounting is similar to the tax treatment described above. In full cost accounting, all the costs described above, as well as dry hole costs, are capitalized and depleted using the cost depletion method described above.

EGO has the capability of handling any of the tax and book treatments of the items described above. It is a simple matter to tell EGO how to treat the various cost items for the purpose of taxation; furthermore, the depletion can be calculated on a per well basis, per prospect basis, or any other cost center basis. EGO allows the user flexibility in choosing the appropriate scheme.

Future versions of EGO will also include full flexibility in handling and reporting information for the books.

Income Taxes

Income taxes are computed as the tax rate multiplied by the taxable income. The taxable income is computed as follows:

$$\begin{aligned} \text{Taxable Income} = & \text{Gross Income} \\ & - \text{Production Taxes} \\ & - \text{Windfall Profit Taxes} \\ & - \text{Operating Costs} \\ & - \text{Interest Payments} \\ & - \text{Expensed Items (Items Not Part of DD \& A)} \\ & - \text{DD \& A for Tax Purposes} \end{aligned}$$

$$\text{Tax} = \text{Tax Rate} * \text{Taxable Income}$$

Escalation

Prices and costs do not stay constant over the years. Thus, any analysis of projected cash flow must take into account price and cost increases. These increases are taken into account by using escalation (which could be just

inflation or some number different than inflation) rates to determine prices and costs in the future. Escalation can be specified as beginning-year, mid-year, end-year, or continuous. For example, if end-year escalation is specified, the price in year 3 will be the price in year 1 multiplied by $(1 + e[1])$ times $(1 + e[2])$, where $e[1]$ is the escalation rate for year 1 and $e[2]$ is the escalation rate for year 2.

EGO has the capability of providing a very flexible price and cost escalation scheme for the user.

Discounting and Internal Rate of Return

The cash flow projections from the production from any well stretch many years into the future. The value of money available now is different from the value of money obtained in the future. This can be easily seen by asking yourself the question of whether you would like \$1,000 today or \$1,000 1 year from today. Thus, because money in the future will be worth somewhat less than money today, for each year in the future there is a discount factor that, when multiplied by the money obtained in that year, determines what the money is worth in today's terms. Typically each company will specify a discount rate (denoted by r) to be applied to compute the discount factors. The simplest scheme is to assume that all cash flow is obtained at the end of a year. The discount factor in year t is then the reciprocal of $(1 + r)$ multiplied by itself $(t - 1)$ times. This is called end-year discounting. This concept of discounting can be used to evaluate the effect of future cash flow conveniently.

By far, the most important measure of the value of future cash flow is the present worth of the cash flow. This is the sum of all future cash flows discounted to the value of the money in current terms. Using end-year discounting as described above, the "present worth" for a stream of cash flow n years into the future is given by

$$\begin{aligned} \text{Present Worth} = & \text{Sum over Years 1 through } n \\ & (\text{Cash Flow in Year } t \\ & * \text{Discount Factor for Year } t) \end{aligned}$$

Usually present worths are computed on both a before tax cash flow and after tax cash flow.

Another concept that is often used in evaluating the worth of an investment is the internal rate of return (denoted by IRR). This is the discount rate that causes the present worth of negative cash flow (investments and costs) to be equal to the present worth of the positive cash flow (revenues generated from the investment). Often, the IRR is also called the yield from the investment. Note that the IRR should be used with caution, because when there is alternating negative and positive cash flow for many years into the future, there may be more than one IRR, in which case the correct way to look at the investment is through the present worth profiles (i.e., present worth obtained for various discount rates).

EGO allows the user to specify up to 25 different discount rates for computing present worth. Furthermore, the user can specify whether the discounting should be beginning-year, mid-year, end-year, or continuous. The IRR routine also uses the method of discounting specified by the user. Furthermore, EGO also provides the user with an indication when multiple IRRs exist.

EVALUATING THE ECONOMICS OF OIL AND GAS RESERVES THROUGH EGO

As mentioned previously, EGO is a computer model designed for the IBM PC/XT/AT, which is extremely flexible, reliable, fast, and easy to use. EGO operates off a data base of wells. The data base is organized by prospects (or fields) that each contain a set of wells. The user can select any combination of wells and/or prospects for the purpose of economic analysis. The user also has the additional flexibility of assigning the prospects to a cost center for accounting purposes. Furthermore, the user can also assign any set of cost centers, prospects, or wells to a region for the purpose of reporting by region.

EGO allows the user flexibility in specifying all the data for the wells. For example, the user can specify any type of production decline curves (including asking the model to fit a particular curve to data. Virtually any data item can be specified in a manner as desired by the user. Among the data items are escalation rates, discount rates, oil prices, gas prices, heat content, drilling costs, lease costs, operating costs, working interests, net revenue interests, reversion point information, and so forth.

EGO allows the user to specify whether to use successful efforts accounting or full cost accounting. In addition, EGO is flexible enough to handle tax/SEC changes. Most of the key computations are defined through user input. For example, the user can specify whether the depletion is to be computed on a well-by-well basis, a prospect basis, or a cost center basis. Thus, in most situations, it is very easy to customize EGO for a company.

EGO was developed after Stanford Business Software, Inc., developed a "Prospect Evaluation Model" (PEM) for Natural Gas Corporation, Inc., of California (NGC). PEM was custom developed and has many features that are specific to the situation at NGC. PEM has been in continuous use at NGC for over 3 years. Although EGO has its roots in PEM, it is a completely new and different version and has been rewritten for enhanced flexibility and speed. However, the concepts used in EGO have been tested in various software packages for over 10 years by the developers of the software.

The EGO development team included numerical analysts; thus, the computations done in EGO are extremely reliable numerically.

EGO has a flexible reporting scheme. It includes a set of predefined report formats that incorporate enough detail for most users. It also includes a report generator that allows users to set up reports according to their tastes. Some graphic reports can also be generated on the screen and on some commonly used plotters. EGO also has the capability of providing standard files that can be read by other software packages supporting extended graphic capabilities, thus providing the user with unlimited graphing potential.

EGO is extremely fast. It typically takes about 2 seconds to perform the computations for each well. This is a small fraction of the amount of time taken by EGO's competitors.

Following is an example of selecting any combination of fields and wells for a particular run.

Example of Selection Process for Region, Cost Centers, Prospects, and Wells

The description below is for a computer with a monochrome monitor. If a color monitor is used, various colors are used to indicate when items are selected and the state of selection (detailed or global) of the item.

The space bar is used to select a region, project letter, prospect, well name, or data item. Depressing the space bar once causes the particular item to be selected. The selected item appears on the screen in highlighted mode. Hitting the space bar a second time (i.e., when an item is already in the selected mode) will cause the item to be selected for detailed selection where applicable. An item selected for detailed selection appears on the screen in highlighted mode with an asterisk on its left. Hitting the space bar a third time (i.e., when an item has been selected for detailed selection) will cause the item to be deselected. If an item cannot be selected for detailed specifications, hitting the space bar when the item has been selected (i.e., when the item is highlighted on the screen) will cause the item to be deselected. An example of an item that cannot be selected in detailed mode is a well. There are function keys available to select (or detail select) a number of items quickly.

The arrow keys on the number keypad allow the user to move from one item to another.

The following example shows how the selection screens look at different stages of the selection process. The selection process is used for global edit, data update, and running the model. The first descriptive line that appears on the computer screen to let the user know which part of the program is being accessed has been left out of the following example.

The symbol >> indicates where the cursor is on the screen. In this example, Region 1 has been selected for detailed selection, and Region 2 is not selected.

```

SELECT REGION
>> *Region 1  Region 2

```

After hitting the Enter (Return) key at the end of the list, the screen shown below is displayed for selection of cost centers from Region 1. From this screen, Cost Center 6 is chosen for detailed selection, and Cost Center 8 is chosen to include all its prospects and wells.

Region: Region 1

```

                SELECT COST CENTERS
Cost Center 1   Cost Center 2   Cost Center 3
Cost Center 4   Cost Center 5   *Cost Center 6
Cost Center 7   Cost Center 8   >> Cost Center 9
Cost Center 10  Cost Center 11

```

After selecting Cost Center 6 and Cost Center 8, depress the End key followed by the Enter key. This causes the program to display the prospects for each of the cost centers chosen for detailed selection. Thus, continuing

our example, suppose that Prospect 2 is selected for detailed selection. The completed prospect selection screen is shown below.

Region: Region 1 Cost Center: Cost Center 6

SELECT PROSPECTS

>> Prospect 1 *Prospect 2

Once Prospect 2 has been selected, the End key is depressed followed by depressing the Enter key. This then brings up the selection screen for wells. From the well selection screen, all the wells are selected by hitting the F1 key. The selection screen is shown below.

Region: Region 1 Cost Center: Cost Center 6 Prospect: Prospect 2

SELECT WELLS

>> Well #1 Well #2

CONCLUSIONS

This article has provided a brief introduction to the issues involved in gas/oil economics. Many details have been omitted as it is not possible to explain the intricacies of oil and gas economics in a short article such as this.

We have also covered key features of EGO, a model to evaluate gas and oil economics, which runs on the IBM PC and is extremely flexible, fast, and easy to use. Typically, EGO can perform the calculations to analyze one well in approximately 2 seconds. The concepts used in EGO have been tested in various software packages for over 10 years by the developers of the software.

ACKNOWLEDGMENTS

I would like to thank Peter Laventhol and G. Sekar for their helpful comments in writing this article and for their involvement in the EGO project. I would also like to thank Myron Jones, Jim Lynch, Scott Leonhard, Rose Gutierrez, and various other staff members of NGC for introducing me to the concepts of oil and gas property evaluation. Finally, I would especially like to thank Theodore Fong, also of NGC, for his many helpful comments in designing the user interface for the PEM, which preceded the development of EGO.

BIBLIOGRAPHY

- Gallun, R. A., and J. W. Stevenson, *Fundamentals of Oil and Gas Accounting*, Penn Well Publishing Company, Tulsa, OK, 1986.
- Klingstedt, J. P., and H. R. Brock, *Oil and Gas Tax Handbook*, Professional Development Institute, Denton, TX, 1986.
- Megill, R. E., *Exploration Economics*, Petroleum Publishing Company, Tulsa, OK, 1979.
- Robuck, I. F., Jr., *Economic Analysis of Petroleum Ventures*, IED Exploration, Inc., Institutes of Energy Development, P.O. Box 45941, Tulsa, OK 74145, 1980.

- Thapa, M. T., and P. Laventhol, *Prospect Evaluation Model—User's Guide*, Stanford Business Software, 2672 Bayshore Parkway, Suite 304, Mountain View, CA 94043, 1986.
- Thapa, M. T., P. Laventhol, G. Sekar, and B. Menon, *Economics of Gas and Oil Reserves (EGO)—User's Guide and Introduction to Oil and Gas Accounting*, Stanford Business Software, 2672 Bayshore Parkway, Suite 304, Mountain View, CA 94043, 1987.
- Thompson, R. S., and J. D. Wright, *Oil Property Evaluation*, Thompson-Wright Associates, P.O. Box 892, Golden, CO 80402, 1985.

MUKUND N. THAPA

THE ANALYTIC HIERARCHY PROCESS AND THE PERSONAL COMPUTER

This article will discuss how the analytical hierarchy process (AHP), when implemented on a microcomputer, produces a man-machine synergy that enables people to make better decisions about complex problems.

INTRODUCTION—MULTICRITERIA DECISION PROBLEMS

What constitutes a complex problem, making a decision difficult? The measure of complexity must be related to the human thought process. The first ingredient of complexity occurs when we are required to choose from a number of alternatives; the greater the number of alternatives, the greater the complexity.

Complex decisions require us to make judgments. In order to make judgments, standards on which judgments can be based are required; the dictionary calls these criteria. Without criteria we would have no basis for judging. The second ingredient of complexity occurs when there are several, often competing, criteria.

We will consider a problem to be complex if it possesses the following two characteristics: (a) numerous (at least two) feasible alternatives and (b) numerous (at least two) criteria or objectives.

The "complex" problems that we will discuss, that is, those having several alternatives and several criteria, have come to be called multicriteria decision problems. The complexity stems not so much from the size of the problem itself but from the fact that multicriteria are often at odds with one another.

According to Cook and Hammond [1], "Reaching a decision will ordinarily involve making trade-offs between the factors affecting the decision, and this is a difficult and poorly understood aspect of decision making. Research conducted over the past quarter century has demonstrated conclusively that people have a poor understanding of how they make such trade-offs."

Jones [2] cites experimental evidence that shows that "... people do find it difficult to make consistent judgments when more than one attribute is involved." He also states that there is "... a strong argument for providing some structured means of helping them 'think through' their choices."

IDENTIFICATION OF ALTERNATIVES, CONSTRAINTS, AND CRITERIA

When confronted with a problem, we consider alternatives and then choose the one (or combination) that best meets our goal. In order to make the

"best" decision, we must first identify alternatives, eliminate those that are not feasible, and then choose according to our criteria.

Identification of Alternatives

Identifying alternatives to a problem is often more of an art than a science. A popular technique often used by a group of people is called brainstorming. The objective of brainstorming is to determine all possible alternatives to a problem, regardless of feasibility or merit.

Identification of Criteria

In order to judge the feasibility and merits of the alternatives to a problem, we must judge how well each of the alternatives satisfies our requirements. Criteria are the standards on which we can make such judgments. Brainstorming can also be used to identify criteria, as well as alternatives.

Kepner and Tregoe [3] have proposed that we consider two classes of criteria: "musts" and "wants." Distinguishing between must and want criteria, although not essential, is often easy to do and simplifies the subsequent analysis. In theory, there is not always a clear distinction between a must and a want. We often say we *must* have something when, in fact, we mean we *want* something very badly. In practice, however, it is often an easy matter to make the distinction between musts and wants.

If it is obvious that an alternative identified during brainstorming is not a viable alternative, by stating why it is not viable, one has identified a must criteria, or constraint. For example, in selecting a mode of transportation to be used on the highways, most adults might have a safety requirement that the vehicle possess more than two wheels, eliminating motorcycles and mopeds.

Must Criteria (Constraints)

Each must criteria is a constraint that, if not met by an alternative, will eliminate that alternative from further consideration. In other words, in order for an alternative to be considered feasible, it must satisfy each and every constraint.

If brainstorming is used to generate alternatives, many, if not most, of the alternatives would not really make much sense. For example, in selecting a vehicle for transportation to and from work, a golf cart might be eliminated because there is no route between home and work on which it would be legal to drive a golf cart. That is, we can identify a constraint that requires that a route exist between home and work that accommodates the vehicle and another constraint that it be legal to operate the vehicle on that route. Any vehicle that does not satisfy either constraint is considered to be infeasible and is eliminated from further consideration.

A question to ask when brainstorming for must criteria is, What is the absolute minimum that an alternative must possess with respect to a certain characteristic?

Wants

Those criteria that do not specify absolute minimums are wants. In practice, most criteria are wants. A question to ask when identifying criteria

classified as wants is, What characteristic might influence me to choose this alternative as opposed to that?

After identifying alternatives and criteria, categorizing the criteria into musts and wants, and then eliminating those alternatives that do not satisfy all of the must criteria, the problem is narrowed to consideration of the remaining alternatives with respect to the want criteria.

The problem of how to best satisfy wants is endemic to every facet of human endeavor and has, until recently, eluded a reasonable solution. The AHP provides a reasonable solution.

COMPLEXITY AND ORGANIZATION

Complex problems require organization. The greater the number of criteria, the greater the complexity is. The greater the number of alternatives, the greater the complexity is. There is, however, a limit to the ability of the mind to relate and discriminate among and to accurately choose from a large number of alternatives, even if there were only one criterion.

According to James Martin [4], if a person "has to choose from a range of 20 alternatives, he will give inaccurate answers because the range exceeds the bandwidth of his channel for perception. In many cases, seven alternatives are the approximate limit of his channel capacity." Martin's conclusion is based on the results of numerous psychological experiments, including the well-known study "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Information Processing," by G. A. Miller [5].

Martin observed that "Seven is a low number when we are concerned with complex problem solving." But he also observed that "Fortunately, there are several ways of increasing our effective channel capacity. The first is to enable the subject to make relative rather than absolute judgments. The second is to organize tasks so that the subject makes several judgments in succession" [4].

Another dimension of problem complexity is related to what Martin refers to as the capacity of the human short-term memory--the number of items that the brain can store for immediate processing and which will lose its contents very quickly if not reinforced. Psychologists have also measured the capacity of the human short-term memory and have concluded that the short-term memory can hold about seven separate items (this varies by plus or minus two for normal subjects in different situations.)

Fortunately, we can accommodate conflicting criteria, as well as compensate for our limited mental channel capacities and short-term memories by organizing a problem into a meaningful structure. What form should this organization take?

HIERARCHICAL ORGANIZATION

Our minds structure complex reality into its constituent parts and these, in turn, into their parts, and so on. This hierarchical organization is a natural thought process. Saaty [6] has pointed out that our tendency to think that hierarchies were invented by corporations and governments is false. Hierarchies are basic to the human way of breaking reality into clusters and subclusters. Whyte [7] expressed this thought as follows:

The immense scope of hierarchical classification is clear. It is the most powerful method of classification used by the human brain-mind in ordering experience, observations, entities, and information.... The use of hierarchical ordering must be as old as human thought, conscious and unconscious...

Herbert Simon, nobel laureate, observed the following:

Large organizations are almost universally hierarchical in structure. That is to say, they are divided into units, which are subdivided into smaller units, which are, in turn, subdivided and so on.... Hierarchical subdivision is not a characteristic that is peculiar to human organizations. It is common to virtually all complex systems of which we have knowledge. . . . The near universality of hierarchy in the composition of complex systems suggests that there is something fundamental in this structural principle that goes beyond the peculiarities of human organization. . . . An organization will tend to assume hierarchical form whenever the task environment is complex relative to the problem-solving and communicating powers of the organization members and their tools. Hierarchy is the adaptive form for finite intelligence to assume in the face of complexity [8].

Saaty describes how theory reflects what appears to be an innate method of operation of the human mind:

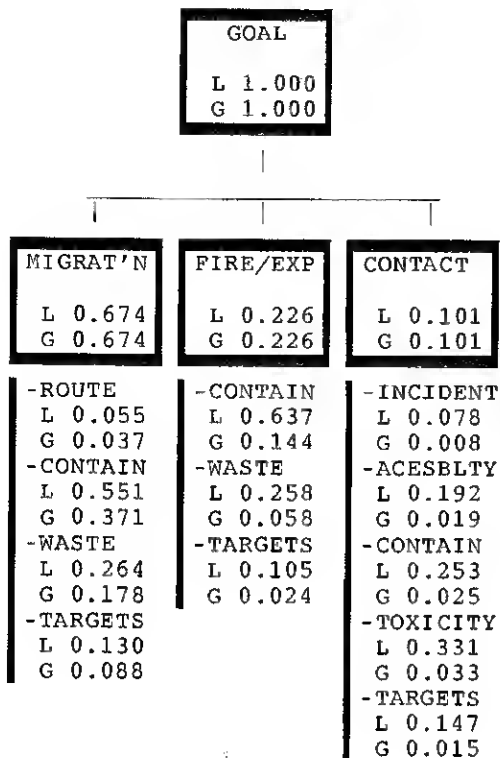
When presented with a multitude of elements, controllable or not, which comprise a complex situation, (the mind) aggregates them into groups, according to whether they share certain properties. Our model of this brain function allows a repetition of this process, in that we consider these groups, or rather their identifying common properties, as the elements of a new level in the system. These elements may, in turn, be grouped according to another set of properties, generating the elements of yet another "higher," level, until we reach a single "top" element, which can often be identified as the goal of our decision-making process [6].

THE AHP

The AHP enables decision makers to visually structure a complex problem in the form of a hierarchy. Each factor and alternative can be identified and evaluated with respect to other related factors. Although there are many variations of a hierarchy, a typical one contains a goal, below which are criteria, below which are levels of subcriteria and, finally, alternatives. Figure 1 illustrates the first three levels of an AHP hierarchy (Fig. 1 will be explained in the case study presented later in this text).

The ability to structure a complex problem and then focus attention on specific components amplifies one's decision-making capabilities. The capacity for making judgments is expanded beyond the limits imposed by, what psychologists call, the limited channel capacity and short-term memory of the human mind.

Once a hierarchy is established, expert judgments are solicited from the decision maker relative to each facet of the decision problem. The method-



GOAL --- Evaluate potential hazards to environment
 ACESBLTY --- Accessibility of hazardous substances
 CONTACT --- Direct contact mode
 CONTAIN --- Containment category
 FIRE/EXP --- Fire and explosion hazard mode
 INCIDENT --- Observed incident
 MIGRAT'N --- Migration hazard mode
 ROUTE --- Route characteristics category
 TARGETS --- Targets category
 TOXICITY --- Toxicity/persistence
 WASTE --- Waste characteristics category

L --- LOCAL PRIORITY: PRIORITY RELATIVE TO PARENT
 G --- GLOBAL PRIORITY: PRIORITY RELATIVE TO GOAL

FIGURE 1 Evaluate potential hazards to environment—GOAL.

ology goes beyond conventional decision analysis techniques by not requiring numerical guesses. Subjective judgments on aspects of a problem for which no scale of measurement exists are easily accommodated.

The expert(s) providing the judgments can do so entirely in a verbal mode; no numerical guesses are required. For example, they can enter

judgments about the importance of criterion *A* versus *B*, where criterion *A* is strongly more important than criterion *B*. Alternatively, the experts can provide judgments in a numerical mode in the form of a ratio scale, where criterion *A* is four times more important than criterion *B*.

The judgments are used in deriving relative priorities for the decision criteria and alternatives. This is accomplished by calculating what mathematicians call eigenvalues and eigenvectors of matrices that represent the pair-wise comparisons of criteria and alternatives. This methodology has been shown to provide remarkably accurate results.

The methodology also develops a measure of inconsistency. This measure is useful in identifying possible errors in judgments, as well as actual inconsistencies in the judgments themselves. The decision maker can request suggestions for improving consistency. It is important to note that the methodology does not preclude inconsistencies in judgments. On the contrary, many decisions must be made recognizing inconsistencies that exist in the real world.

After the decision maker conveys his expert judgments relative to the many facets of the decision being contemplated, the results are synthesized into overall priorities. In addition to developing the overall priorities, each of the components that contributes to these overall priorities is presented so that the decision maker can assess the results, implementing them if they seem appropriate or refining the model to reflect additional insights that have been acquired.

Absolute vs. Relative Judgments—Pair-Wise Comparisons

There are two basic kinds of judgments: absolute and relative. In making an absolute judgment, we compare one "thing" against a recollection from our long-term memory. In making a relative judgment, we make a pair-wise comparison of two things, one to the other. Humans can make much better relative judgments than absolute judgments because they have the ability to discriminate between the members of a pair by judging the intensity or preference for one over the other with respect to some criterion.

Pair-wise comparisons are as natural for humans as binary counting is for computers. The AHP uses pair-wise comparisons, as have many other judgment schemes in the past. What makes AHP so different and effective is that (a) judgments can be made verbally (because numerical guesses are not required, subjective judgments can be accommodated), and (b) the mathematical analysis performed on the judgments produces results that are amazingly accurate and robust and that include a measure of the consistency of the judgments.

People are generally wary, if not distrustful, when numbers are introduced into the traditional process of decision making. In some situations, however, appropriately chosen numbers can represent variations in feelings more faithfully than words or rhetoric can.

Two diametrically opposite schools of thought about numbers are (a) that numbers precisely represent quantitative characteristics and (b) that numbers cannot represent qualitative characteristics.

Each of these ideas is too extreme. Numbers neither represent quantitative characteristics to humans in a manner as precise as we think, nor are numbers completely out of place when we deal with qualitative or subjective characteristics. Numbers are sometimes artifacts that give us the illusion of greater precision than we may be capable of feeling.

The use of numbers can be extended in some reasonable, easily understandable way to reflect our feelings on various social, economic, and political matters.

Scales of Measurement

We cannot measure without a scale, but traditional scales such as time and money limit the nature of ideas we can deal with. Thus, we need "new" scales.

Should we be afraid to develop new scales? To answer this question, let us consider the history of some of our well-accepted scales.

We should not think of our existing scales as being sacrosanct. In fact, existing scales were developed in an attempt to "quantify" what was, at the time, qualitative or subjective. Scales that are precise today, based upon international standards, were once imprecise and intended to represent and communicate qualitative or subjective characteristics.

Consider the brightness and color of light. Are these qualities quantitative or subjective? For most of our day-to-day purposes, the answer is subjective. Yet we do have precise scales and devices to measure both the brightness (e.g., footcandles) and color (e.g., hertz) of light.

How about time? Many of us now use digital watches that are accurate to within a few seconds a month. But, for most of our purposes, such as judging the amount of time remaining in our lunch "hour," we probably get more useful information by looking at the minute hand of an analog clock than the digits on a digital clock.

To most of us, temperature is more subjective than time. We don't usually wear thermometers and often say things like "Sure feels hot today"! Yet we do have precise scales and devices to measure temperature, as well. One of these devices, the mercury thermometer, was invented by Gabriel Fahrenheit in the 18th century. This scale was based on what was, at the time, subjective. Normal body temperature was arbitrarily assigned a value of 100 degrees (100°F) (later found to be 98.6°F), and the temperature of the coldest thing Fahrenheit could produce in his laboratory, a mixture of salt and ice, was assigned a value of 0°F.

Even when we have precise quantitative measurements, we must provide some subjective interpretations when making decisions. For example, 20°F is not twice as warm as 10°F. And even if we were to use the "absolute" scale, the relationship between the amount of heat in the atmosphere and a human's comfort is not linear. We try to model our subjective feelings by defining other "models," such as wind-chill factors and temperature-humidity indices.

What can executive decision makers do to "feel" for the right decision? What scales can they use to measure corporate image and employee morale? Almost every decision is strongly influenced by subjective criteria and judgments (including politics).

Just as scales were developed in the past, new scales will be developed in the future. As such scales are developed, our notions of what is objective and what is subjective will change again. There is no reason for us to be afraid to develop or use new scales.

Natural Ratio Scale

A "natural" scale used by the AHP consists of forming the ratio of two things. For example, when comparing the weights of two objects, they may

be equal or, perhaps, one may be twice as heavy as the other. The scale is natural in the sense that we do not have to measure against some standard, such as a pound or a kilogram. In comparing the brightness of two lights, we may judge that one is three times as bright as the other, without resorting to measurements against a standard such as footcandles.

The scale is universal in that it can be used to compare anything with respect to a characteristic of interest. It can even be used to compare apples and oranges! With respect to weight, an apple and an orange are about equal. With respect to size, an apple and an orange are about equal. With respect to brightness, an orange is brighter than an apple. How much brighter? Twice as bright? Three times as bright? Can we make such a judgment without a light meter? Of course! Will such judgments be accurate? In the sense of accurately reflecting your subjective feeling, the answer is yes, by definition. What about in the sense of agreeing with measurements made with a light meter?

Modes of Comparison

Saaty [9] has demonstrated the feasibility of expressing the relative importance of one element over another with respect to a given criterion either verbally or numerically. Table 1 illustrates both modes.

On the numerical scale, 1.0 implies that the elements are equally important, 2.0, that one element is twice as important as the other, and 9.0, that one element is nine times as important as the other. Experience has confirmed that a scale of nine units is reasonable and reflects the degree to which we can discriminate the intensity of relationships among elements.

When making comparisons in a social, psychological, or political context, it is usually more appropriate to make verbal comparisons. When comparing economic or other measurable factors, numerical comparisons may be more appropriate (however, not always, because we have a tendency to overlook the nonlinearities in our utility when we use such numbers directly.)

Almost every decision involves one or more criteria for which no quantifiable scale exists. In fact, the higher the level of decision making within an organization, the more important these nonquantifiable criteria become and the greater the benefit of AHP. The natural ratio scale and the verbal comparisons can be meaningfully applied to criteria that are typically thought of as being nonquantifiable and subjective.

Redundant Judgments

When making judgments in the form of pair-wise comparisons, the relative importance of n factors can be arrived at by making only $(n - 1)$ judgments. For example, if we were comparing the relative importance of three factors, A , B , and C , and we judged that A is three times more important than B and that A is six times more important than C , we could surmise that B is three times as important as C . However, each of our two judgments might contain some error. The common statistical approach to minimizing errors is to take additional observations. The AHP does something similar by asking the decision maker(s) to make judgments about all possible pairs of $n*(n - 1)/2$ of factors, instead of the minimum $(n - 1)$ judgments.

Thus, in the above example, when asked to judge the relative importance of B and C , we might say that B is two times more important than

TABLE 1 Verbal and Numerical Judgment

Numerical Scale	Verbal Scale	Explanation
1.0	Equal importance of both elements	Two elements contribute equally to the property
3.0	Moderate importance of one element over another	Experience and judgment favor one element over another
5.0	Strong importance of one element over another	An element is strongly favored
7.0	Very strong importance of one element over another	An element is very strongly dominant
9.0	Extreme importance of one element over another	An element is favored by at least an order of magnitude of difference
2.0, 4.0, 6.0, 8.0	Intermediate values between two adjacent judgments	Used for compromise between two judgments
Increments of 0.1	Intermediate values in increments of 0.1 (Example: 6.3 is a permissible entry)	Used for even finer gradations of judgments

C. This judgment, redundant in the sense that we could have calculated that *B* was three times more important than *C* from the first two judgments, may be more accurate than either of the first two judgments. Thus, by using judgments about all possible pairs, the effects of errors in judgments can be minimized by "averaging." In addition, a measure of inconsistency can be derived from the set of pair-wise comparisons.

Calculating Priorities

The mathematical calculation of priorities derived from pair-wise comparisons with the AHP does indeed reduce the effects of errors in judgments. Although the details are somewhat involved, the results are impressively accurate.

The details involve the calculation of eigenvalues and eigenvectors of reciprocal matrices formed from the decision makers' judgments (see Saaty [9]). The process is easily understood by engineers and mathematicians, although the many calculations for a complex problem are usually laborious. Fortunately, the process can be implemented on a personal computer so that a decision maker is not the least concerned with the theory or the complexity of the calculations.

Can we use the process to compare subjective qualities, such as taste? Certainly! I may prefer apples twice as much as oranges, and you may

prefer oranges twice as much as apples. Of course, there is no "correct" judgment. Each of us is the "expert" when making such judgments, and judgments of this type are implicit in almost every high-level decision-making process. Fortunately, the AHP now enables us to measure intangible, subjective qualities.

Consistency

A very interesting and important aspect of the AHP approach to deriving priorities is that a measure of inconsistency follows from the calculations performed on the pair-wise judgments. This measure, called a consistency index by Saaty [9], is zero when all judgments are perfectly consistent with one another and becomes larger, as the inconsistency becomes greater. In order to interpret the meaning of a large index, Saaty simulated a very large number of random pair-wise comparisons for different size matrices, calculating the consistency indices and arriving at an average consistency index for random judgments. He then defined the consistency ratio as the ratio of the consistency index for a particular set of judgments to the average consistency index for random comparisons for a matrix of the same size.

Because a set of perfectly consistent judgments produces a consistency index of zero, the consistency ratio will also be zero. A consistency ratio of one indicates consistency akin to that which would be achieved if judgments were not made intelligently but, rather, at random.

Saaty states that it is reasonable, even desirable to allow for some inconsistency; for if we demanded perfect consistency in all of our judgments, we would be dogmatic to the point of refusing to entertain new notions or ideas. From a practical point of view, we know that the real world is very often inconsistent. It is not exceptionally rare in professional sports, for example, for team A to have a better record against team B, for team B to have a better record against team C, and for team C to have a better record against team A.

Saaty argues that a consistency ratio of 0.1 or less (i.e., a consistency index of 10% or less than that obtained from random judgments) is reasonable.

It may be that an inconsistent judgment is due to a clerical error, such as pressing the wrong key on the computer keyboard or entering just the inverse of what the decision maker really had in mind. Such clerical errors are easy to identify and correct. It may be that the decision maker's mind wandered a bit when making the judgments. Or it may be that the decision maker perceives an inconsistency in the real world, in which case the judgments would be left as is.

It is important to emphasize that the objective is to make "good" decisions, not to minimize the inconsistency ratio. Good decisions are most often based on consistent judgments, but the converse is not necessarily true. It is easy to make perfectly consistent judgments that are nonsensical and result in terrible decisions.

Validation

To be credible, this new approach (the natural pair-wise ratio scale and the mathematical technique used to average judgments) should work in areas where we already know the unit of measurement. In fact, it has been vali-

TABLE 2 Light Source Experiment

Chair	Estimate Experiment Group 1	Estimate Experiment Group 2	Results for Applying In- verse Square Law
1	0.61	0.62	0.61
2	0.24	0.22	0.22
3	0.10	0.10	0.11
4	0.05	0.06	0.06

dated in hundreds of experiments that the method does indeed generate results conforming to classic ratio scale measurement in physics, economics, and other fields where standard measures already exist.

In one experiment conducted by Saaty, chairs were placed at various distances (9, 15, 21, and 28 yards) from a light source to see if people, standing at the light source, could judge the relative brightness of the chairs. The experiment was conducted with two groups of people. Neither group was familiar with the inverse square law of optics, which says that relative brightness is inversely proportional to the square of the distance of the object from the light source. The results achieved with the AHP approach were in very close agreement with the inverse square law, as can be seen in Table 2.

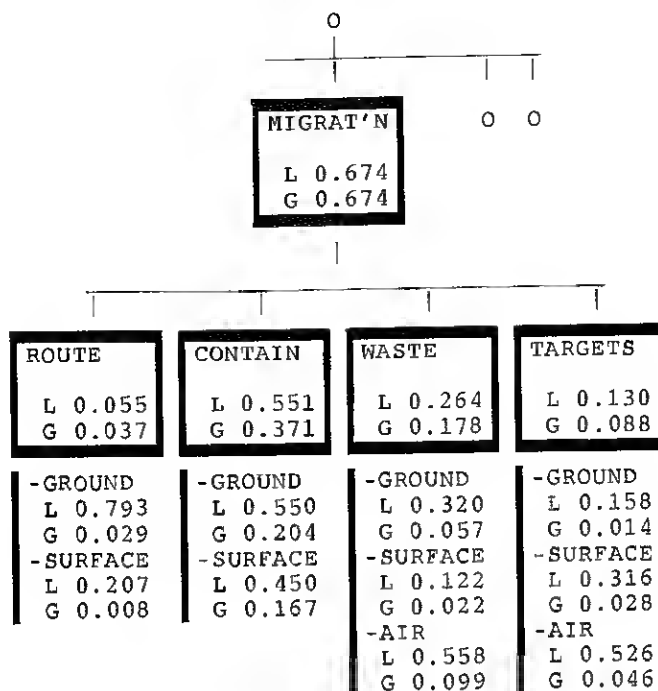
Synthesis

An important contribution of AHP is that it enables decision makers to structure complex problems in the form of a hierarchy and concentrate their attention on each part of the hierarchy, in turn. Judgments can be made about the importance of criteria relative to the goal; then about the importance of each set of subcriteria relative to the criteria; then about the relative importance of the sub-subcriteria relative to each subcriteria, and so on, until judgments are made about the relative preference of alternatives with respect to the lowest level of subcriteria.

Once this is done, the results must be synthesized to arrive at an overall solution. The synthesis used in AHP will be illustrated in the following case study.

CASE STUDY

The Mitre Corporation, acting as a contractor for Environmental Protection Agency, was assigned the task of evaluating potential hazards to the environment. The results of their study were published in the Federal Register on July 16, 1982. Although this study was performed prior to the development of Expert Choice, a microcomputer software adaptation of AHP [10], we will use the criteria, subcriteria, and alternatives to illustrate how well suited AHP is to complex real-world problems.



AIR --- Air route
 CONTAIN --- Containment category
 GROUND --- Ground water route
 MIGRAT'N --- Migration hazard mode
 ROUTE --- Route characteristics category
 SURFACE --- Surface water route
 TARGETS --- Targets category
 WASTE --- Waste characteristics category

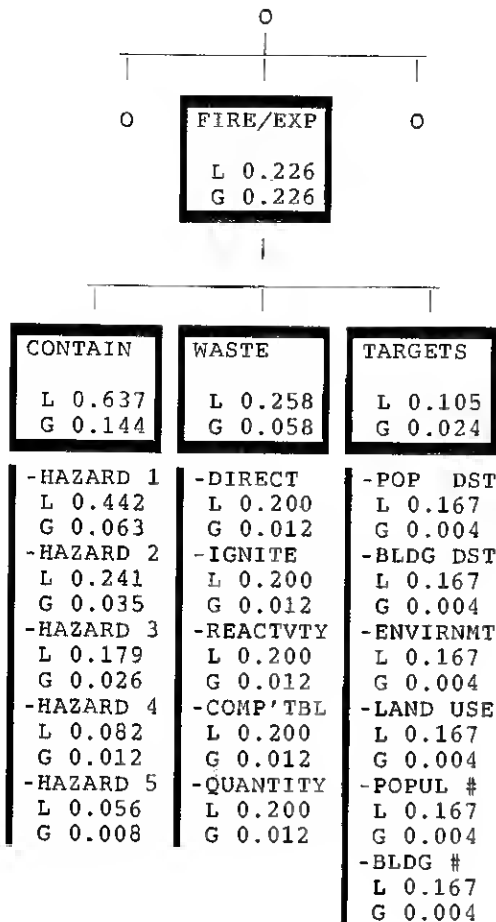
L --- LOCAL PRIORITY: PRIORITY RELATIVE TO PARENT
 G --- GLOBAL PRIORITY: PRIORITY RELATIVE TO GOAL

FIGURE 2 Evaluate potential hazards to the environment—MIGRATION.

Expert Choice Model

Figure 1 illustrates an Expert Choice model to evaluate potential hazards to the environment. This model has three main criteria representing the mode of the potential hazard: migration, fire and exposure hazard mode, and direct contact. Each of these criteria have subcriteria. Figure 1 illustrates the first three levels of the hierarchy. Using Expert Choice (EC) terminology, it shows the first three levels of the EC tree. The goal is at level 0, the criteria are at level 1, and the subcriteria are at level 2.

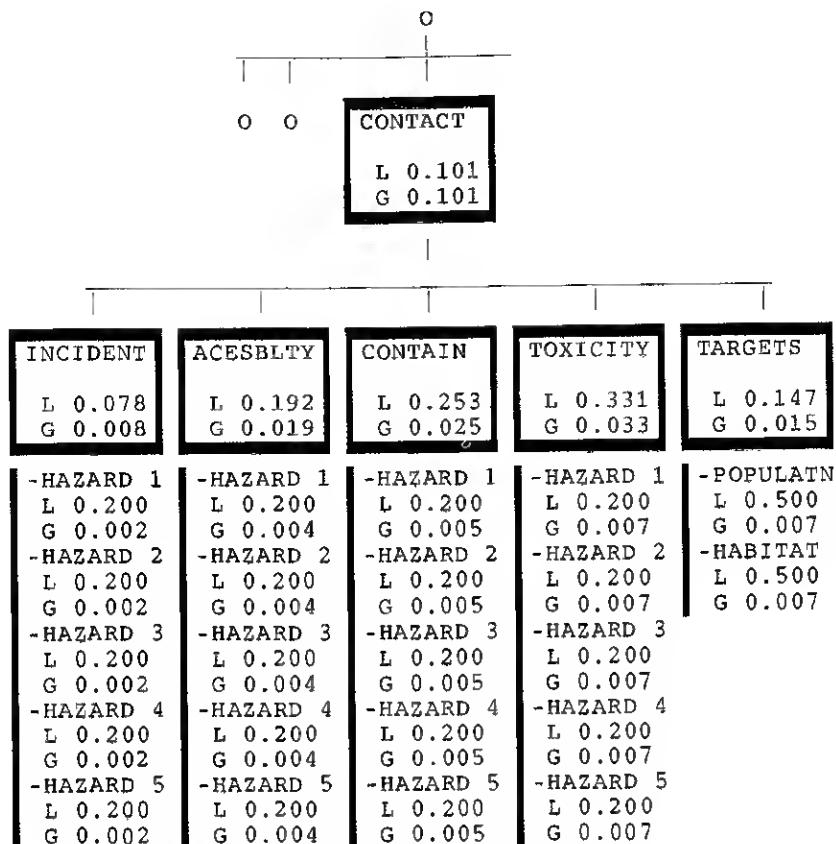
Each element in the EC tree is called a node. The numbers next to the "L" in each node of the model are called "local" priorities and represent the weights derived from the decision makers' judgments. These judgments are entered into the model using either EC's verbal or numerical compari-



BLDG # --- Buildings within 2 mile radius
 BLDG DST --- Distance to nearest building
 COMP'TBL --- Incompatibility with other wastes at site
 CONTAIN --- Containment category
 DIRECT --- Direct evidence
 ENVIRNMT --- Distance to sensitive environment
 FIRE/EXP --- Fire and explosion hazard mode
 IGNITE --- Ignitability
 LAND USE --- Land use
 POP DST --- Distance to nearest population
 POPUL # --- Population within 2 mile radius
 QUANTITY --- Hazardous waste quantity
 REACTVITY --- Reactivity/Incompatibility
 TARGETS --- Targets category
 WASTE --- Waste characteristics category

L --- LOCAL PRIORITY: PRIORITY RELATIVE TO PARENT
 G --- GLOBAL PRIORITY: PRIORITY RELATIVE TO GOAL

FIGURE 3 Evaluate potential hazards to the environment --FIRE/EXP.



ACESBLTY --- Accessibility of hazardous substances
 CONTACT --- Direct contact mode
 CONTAIN --- Containment category
 HABITAT --- Distance to a critical habitat
 INCIDENT --- Observed incident
 POPULATN --- Population within 1 mile radius
 TARGETS --- Targets category
 TOXICITY --- Toxicity/persistence

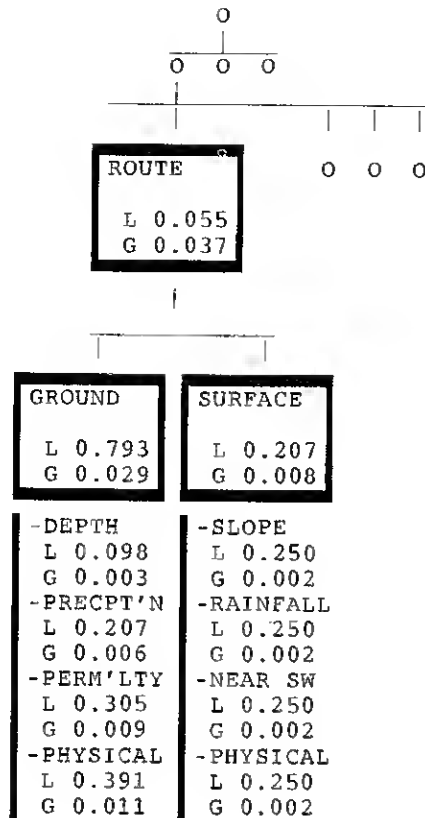
L --- LOCAL PRIORITY: PRIORITY RELATIVE TO PARENT
 G --- GLOBAL PRIORITY: PRIORITY RELATIVE TO GOAL

FIGURE 4 Evaluate potential hazards to the environment—CONTACT.

son modes, as will be shown shortly. The glossary at the bottom of the figure contains the meanings of the node names.

The numbers next to the "G" in each node are called "global" priorities and are found by taking the product of the local priority of a node by the global priority of its parent node.

An EC tree can be redrawn about any node to show lower levels of the model. Figure 2 shows the model redrawn about the migration node.



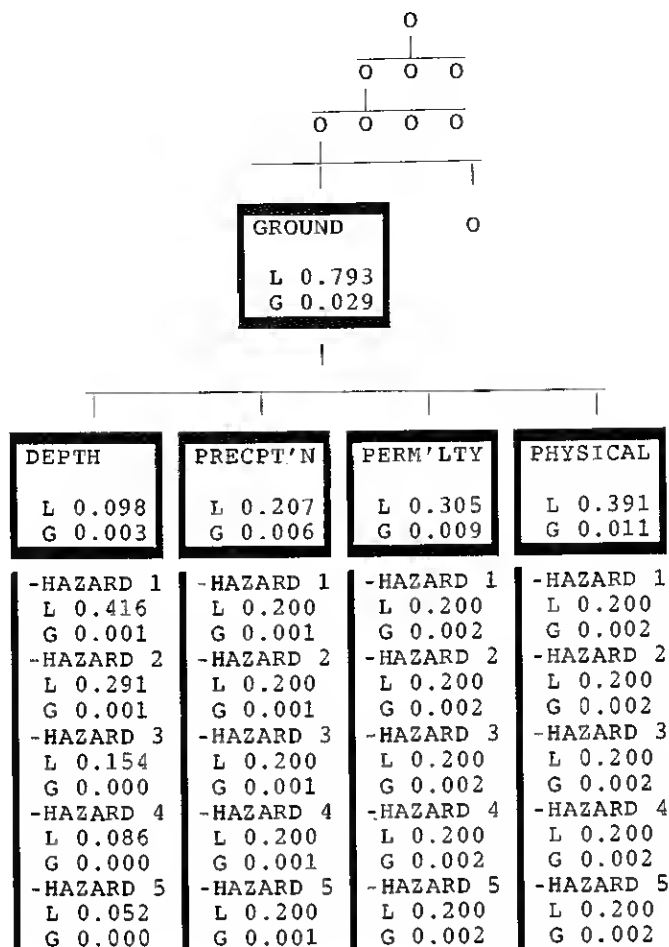
DEPTH --- Depth
 GROUND --- Ground water route
 NEAR SW --- Near surface water
 PERM'LTY --- Permeability
 PHYSICAL --- Physical state
 PRECPT'N --- Net precipitation
 RAINFALL --- Rainfall
 ROUTE --- Route characteristics category
 SLOPE --- Facility slope and intervening terrain
 SURFACE --- Surface water route

L --- LOCAL PRIORITY: PRIORITY RELATIVE TO PARENT
 G --- GLOBAL PRIORITY: PRIORITY RELATIVE TO GOAL

FIGURE 5 Evaluate potential hazards to the environment—ROUTE.

Notice that the siblings and ancestry of the migration node are shown in skeletal form. Also notice that each subcriterion has other factors or sub-criteria of its own. For example, the waste subcriterion has ground, surface, and air factors (level 4 of the tree).

Figure 3 shows the model redrawn about the fire and exposure mode node. Notice the subcriteria appearing below the waste and targets nodes.



DEPTH --- Depth
 GROUND --- Ground water route
 PERM'LTY --- Permeability
 PHYSICAL --- Physical state
 PRECPT'N --- Net precipitation

L --- LOCAL PRIORITY: PRIORITY RELATIVE TO PARENT
 G --- GLOBAL PRIORITY: PRIORITY RELATIVE TO GOAL



FIGURE 6 Evaluate potential hazards to the environment--GROUND.

Because the containment node does not have any criteria, the alternative hazards that are to be evaluated appear below the containment node. If we were to look further down below the waste or target nodes, we would find these same alternatives. Thus, the EC tree model need not be "balanced."

	MIGRAT'N	FIRE/EXP	CONTACT
MIGRAT'N		4.0	5.0
FIRE/EXP			3.0
CONTACT			

Matrix entry indicates that ROW element is
 1 EQUALLY 3 MODERATELY 5 STRONGLY 7 VERY STRONGLY 9 EXTREMELY
 more IMPORTANT than COLUMN element
 unless enclosed in parenthesis.

MIGRAT'N :Migration hazard mode
 FIRE/EXP :Fire and explosion hazard mode
 CONTACT :Direct contact mode

0.674
 MIGRAT'N 
 0.226
 FIRE/EXP 
 0.101
 CONTACT 

INCONSISTENCY RATIO = 0.074

FIGURE 7 Priorities with respect to GOAL to evaluate potential hazards to environment.

Figures 4 through 6 contain more views of this model to evaluate environmental hazards. Although this model is rather detailed, most complex problems are modeled using only two or three levels below the goal.

Judgments and Priorities

Figures 7 through 9 show some of the judgments made with this model. Each entry in the matrix represents a judgment (about the row/column items) entered by the decision maker(s) (see glossary in Fig. 7 for meaning of entries). Typically, judgments are entered in a verbal mode, but displaying judgments in this numerical fashion is more concise.

Figure 10(A, B) shows the results of the synthesis, and Figure 11 shows the final results—hazard 2 was determined to be the most detrimental attribute to the environment.

A histogram of the weights or priorities derived from the judgments follows the judgment matrix. Each figure also shows the inconsistency ratio. Because the larger the value of this ratio, the more inconsistent are the judgments, EC refers to this ratio as the "inconsistency ratio." The value of this inconsistency ratio for the judgments about the relative importance of the main criteria is .074, as shown in Figure 7. This is certainly less than 0.1 and is quite acceptable. If it had been greater than 0.1, one would usually ask EC to identify which judgments were most inconsistent.

	ROUTE	CONTAIN	WASTE	TARGETS
ROUTE		(6.0)	(5.0)	(4.0)
CONTAIN			3.0	5.0
WASTE				3.0
TARGETS				


Matrix entry indicates that ROW element is _____
 1 EQUALLY 3 MODERATELY 5 STRONGLY 7 VERY STRONGLY 9 EXTREMELY
 more IMPORTANT than COLUMN element
 unless enclosed in parenthesis.

ROUTE :Route characteristics category
 CONTAIN :Containment category
 WASTE :Waste characteristics category
 TARGETS :Targets category

0.055
 ROUTE




0.551
 CONTAIN



0.264
 WASTE



0.130
 TARGETS



INCONSISTENCY RATIO = 0.080

FIGURE 8 Judgments and priorities with respect to MIGRATION < GOAL.

Synthesis


Figure 10(A, B) shows the details and the synthesis, and Figure 11 the final results. The synthesis details are sorted by priority, so that the most important factors and subfactors are listed first. The numbers in the synthesis represent the global priorities. For each alternative, the sum of the global priorities with respect to the lowest level subcriteria is the overall priority for that alternative, as shown in a histogram in Figure 11. For example, the overall priority of hazard 2, the hazard that received the highest overall priority in this model, is as follows:


	0.110	with respect to MIGRATION < CONTAIN < GROUND
plus	0.033	with respect to MIGRATION < CONTAIN < SURFACE
plus	0.035	with respect to MIGRATION < WASTE < AIR
plus	0.003	< TOXICITY
etc.	0.003	with respect to MIGRATION < WASTE < AIR
		< REACTIVITY
	0.526	


	TOXICITY	QUANTITY	REACTVITY
TOXICITY		5.0	4.0
QUANTITY			(3.0)
REACTVITY			

Matrix entry indicates that ROW element is _____
 1 EQUALLY 3 MODERATELY 5 STRONGLY 7 VERY STRONGLY 9 EXTREMELY
 more IMPORTANT than COLUMN element
 unless enclosed in parenthesis.

TOXICITY : Toxicity/persistence
 QUANTITY : Hazardous waste quantity
 REACTVITY : Reactivity/Incompatibility

0.674
 TOXICITY 

0.101
 QUANTITY 

0.226
 REACTVITY 

INCONSISTENCY RATIO = 0.074

FIGURE 9 Judgments and priorities with respect to AIR < WASTE
 < MIGRATION < GOAL.

COMMUNICATING RESULTS

Communicating the rationale for a decision about a complex problem is an important problem in and of itself. The AHP results are ideally suited for such communication.

Consider a problem where a consultant performs the analysis and makes a recommendation to a client. Without the use of AHP, a typical consultant's report would consist of appendices containing the pertinent information, the body of the report, and an executive summary of one or two pages. The executive summary is in recognition that the client will not read the entire report, much less the appendices. However, the client does want to examine some of the detail in the report and the appendices. The problem is that there is no way of anticipating what detail the client will be interested in seeing.

In contrast to this reporting mechanism, the AHP can provide an indexed "road map" of the analysis leading up to the recommendation. The results of a decision analysis using AHP contains information about

- What criteria were used in making the recommendation.
- Which criteria were considered most important.
- What likelihoods were attributed to uncertainties.
- Which alternatives were considered preferable, relative to each criterion.

	ROUTE	CONTAIN	WASTE	TARGETS
ROUTE		(6.0)	(5.0)	(4.0)
CONTAIN			3.0	5.0
WASTE				3.0
TARGETS				


Matrix entry indicates that ROW element is _____
 1 EQUALLY 3 MODERATELY 5 STRONGLY 7 VERY STRONGLY 9 EXTREMELY
 more IMPORTANT than COLUMN element
 unless enclosed in parenthesis.

ROUTE :Route characteristics category
 CONTAIN :Containment category
 WASTE :Waste characteristics category
 TARGETS :Targets category

0.055
 ROUTE




0.551
 CONTAIN



0.264
 WASTE



0.130
 TARGETS



INCONSISTENCY RATIO = 0.080

FIGURE 8 Judgments and priorities with respect to MIGRATION < GOAL.

Synthesis


Figure 10(A, B) shows the details and the synthesis, and Figure 11 the final results. The synthesis details are sorted by priority, so that the most important factors and subfactors are listed first. The numbers in the synthesis represent the global priorities. For each alternative, the sum of the global priorities with respect to the lowest level subcriteria is the overall priority for that alternative, as shown in a histogram in Figure 11. For example, the overall priority of hazard 2, the hazard that received the highest overall priority in this model, is as follows:


	0.110	with respect to MIGRATION < CONTAIN < GROUND
plus	0.033	with respect to MIGRATION < CONTAIN < SURFACE
plus	0.035	with respect to MIGRATION < WASTE < AIR
plus	0.003	< TOXICITY
etc.	0.003	with respect to MIGRATION < WASTE < AIR
		< REACTIVITY
	0.526	


	TOXICITY	QUANTITY	REACTVITY
TOXICITY		5.0	4.0
QUANTITY			(3.0)
REACTVITY			

Matrix entry indicates that ROW element is
 1 EQUALLY 3 MODERATELY 5 STRONGLY 7 VERY STRONGLY 9 EXTREMELY
 more IMPORTANT than COLUMN element
 unless enclosed in parenthesis.

TOXICITY : Toxicity/persistence
 QUANTITY : Hazardous waste quantity
 REACTVITY : Reactivity/Incompatibility

0.674
 TOXICITY 

0.101
 QUANTITY 

0.226
 REACTVITY 

INCONSISTENCY RATIO = 0.074

FIGURE 9 Judgments and priorities with respect to AIR < WASTE
 < MIGRATION < GOAL.

COMMUNICATING RESULTS

Communicating the rationale for a decision about a complex problem is an important problem in and of itself. The AHP results are ideally suited for such communication.

Consider a problem where a consultant performs the analysis and makes a recommendation to a client. Without the use of AHP, a typical consultant's report would consist of appendices containing the pertinent information, the body of the report, and an executive summary of one or two pages. The executive summary is in recognition that the client will not read the entire report, much less the appendices. However, the client *does* want to examine some of the detail in the report and the appendices. The problem is that there is no way of anticipating what detail the client will be interested in seeing.

In contrast to this reporting mechanism, the AHP can provide an indexed "road map" of the analysis leading up to the recommendation. The results of a decision analysis using AHP contains information about

- What criteria were used in making the recommendation.
- Which criteria were considered most important.
- What likelihoods were attributed to uncertainties.
- Which alternatives were considered preferable, relative to each criterion.

LEVEL 1	LEVEL 2	LEVEL 3	LEVEL 4	LEVEL 5
-----	-----	-----	-----	-----
MIGRAT'N =0.674				
.	CONTAIN =0.371	GROUND =0.204	HAZARD 2 =0.110	
.	.	.	HAZARD 1 =0.042	
.	.	.	HAZARD 4 =0.033	
.	.	.	HAZARD 3 =0.017	
.	.	.	HAZARD 5 =0.002	
.	.	SURFACE =0.167	HAZARD 3 =0.067	
.	.	.	HAZARD 5 =0.034	
.	.	.	HAZARD 2 =0.033	
.	.	.	HAZARD 1 =0.018	
.	.	.	HAZARD 4 =0.016	
.	WASTE =0.178	AIR =0.099	TOXICITY =0.067	HAZARD 2 =0.035
.	.	.	.	HAZARD 1 =0.014
.	.	.	.	HAZARD 3 =0.009
.	.	.	.	HAZARD 4 =0.005
.	.	.	.	HAZARD 5 =0.004
.	.	.	REACTVITY =0.022	HAZARD 4 =0.006
.	.	.	.	HAZARD 5 =0.005
.	.	.	.	HAZARD 3 =0.005
.	.	.	.	HAZARD 2 =0.003
.	.	.	.	HAZARD 1 =0.003
.	.	.	QUANTITY =0.010	HAZARD 1 =0.002
.	.	.	.	HAZARD 2 =0.002
.	.	.	.	HAZARD 3 =0.002
.	.	.	.	HAZARD 4 =0.002
.	.	.	.	HAZARD 5 =0.002
.	.	GROUND =0.057	TOXICITY =0.028	HAZARD 1 =0.006
.	.	.	.	HAZARD 2 =0.006
.	.	.	.	HAZARD 3 =0.006
.	.	.	.	HAZARD 4 =0.006
.	.	.	.	HAZARD 5 =0.006
.	.	.	QUANTITY =0.028	HAZARD 1 =0.006
.	.	.	.	HAZARD 2 =0.006
.	.	.	.	HAZARD 3 =0.006
.	.	.	.	HAZARD 4 =0.006
.	.	.	.	HAZARD 5 =0.006
.	.	SURFACE =0.022	TOXICITY =0.011	HAZARD 1 =0.002
.	.	.	.	HAZARD 2 =0.002
.	.	.	.	HAZARD 3 =0.002

FIGURE 10 Evaluate potential hazards to environment—
Tally for synthesis of leaf nodes with respect to GOAL.

LEVEL 1	LEVEL 2	LEVEL 3	LEVEL 4	LEVEL 5
				HAZARD 4 =0.002
				HAZARD 5 =0.002
			QUANTITY =0.011	
				HAZARD 1 =0.002
				HAZARD 2 =0.002
				HAZARD 3 =0.002
				HAZARD 4 =0.002
				HAZARD 5 =0.002
	TARGETS =0.088			
		AIR =0.046		
			USE =0.015	
				HAZARD 1 =0.003
				HAZARD 2 =0.003
				HAZARD 3 =0.003
				HAZARD 4 =0.003
				HAZARD 5 =0.003
			D ENV. =0.015	
				HAZARD 1 =0.003
				HAZARD 2 =0.003
				HAZARD 3 =0.003
				HAZARD 4 =0.003
				HAZARD 5 =0.003
			POPULATN =0.015	
				HAZARD 1 =0.003
				HAZARD 2 =0.003
				HAZARD 3 =0.003
				HAZARD 4 =0.003
				HAZARD 5 =0.003
		SURFACE =0.028		
			USE =0.009	
				HAZARD 1 =0.002
				HAZARD 2 =0.002
				HAZARD 3 =0.002
				HAZARD 4 =0.002
				HAZARD 5 =0.002
			D ENV. =0.009	
				HAZARD 1 =0.002
				HAZARD 2 =0.002
				HAZARD 3 =0.002
				HAZARD 4 =0.002
				HAZARD 5 =0.002
			POPULATN =0.009	
				HAZARD 1 =0.002
				HAZARD 2 =0.002
				HAZARD 3 =0.002
				HAZARD 4 =0.002
				HAZARD 5 =0.002
		GROUND =0.014		
			WATERUSE =0.007	
				HAZARD 1 =0.001
				HAZARD 2 =0.001
				HAZARD 3 =0.001

FIGURE 10 (continued)

OVERALL INCONSISTENCY INDEX = 0.03

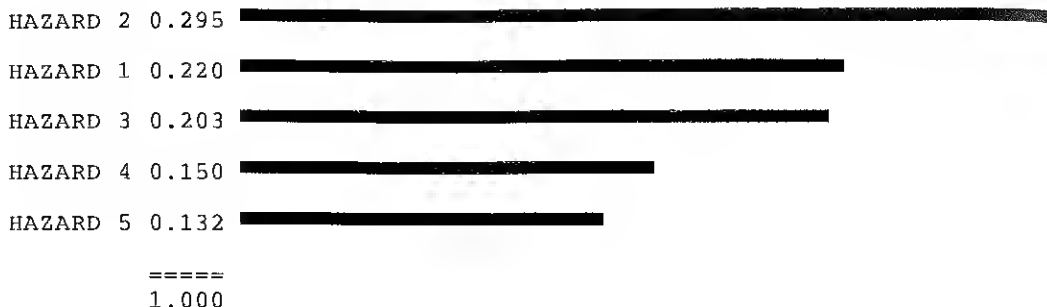


FIGURE 11 Evaluate potential hazards to environment—Synthesis of leaf nodes with respect to GOAL.

The client can then easily pinpoint the most critical or questionable priorities, likelihoods, or preferences and ask for more detail. This detail is at hand in the form of the pair-wise comparisons for the relevant aspect of the problem being questioned.

Finally, the client may ask, Why did you make this particular judgment? The consultant can then either refer to the proper background information contained in one of the appendices or remind the client that the consultant was hired as an expert and that was his or her expert judgment.

SUMMARY

The AHP, with its microcomputer implementation called Expert Choice, combine to provide decision makers with "an expert support system" that

- Is capable of supporting decisions involving several alternatives.
- Is capable of addressing several (often competing) criteria or factors.
- Allows the decision maker(s) to incorporate subjective as well as objective factors.
- Is capable of accommodating the decision makers' expert judgments about the relative importance of these factors.
- Incorporates the decision makers' expertise.
- Allows the decision maker to synthesize or combine judgments made relative to the many facets of a complex problem.
- Does *not* make, or pretend to make, any decisions. (The decision maker(s) must be able to structure the problem as they see it, provide *their* judgments, request the decision support system to help synthesize their judgments, examine the results of the synthesis, restructure the problem if they think it necessary, and eventually arrive at *their* decision.)
- Supports group decision making.
- Has a theoretically sound foundation.

In his early work, called *The New Science of Management Decision*, Herbert Simon observed that "...decision making involves three principal phases: finding occasions for making a decision; finding possible courses of action; and choosing among courses of action." Simon called these three phases intelligence, design, and choice [8]. Executive decisions are unstructured decisions in that all three phases are not fully understood and therefore cannot be programmed.

Luconi et al. [11] have identified the limits of current expert systems and decision support systems technology. The domain of decision support systems does not encompass the strategic aspect (principally choice) of decisions, whereas the knowledge that can be feasibly encoded in an expert system is not sufficient to make satisfactory decisions by itself. Luconi et al. [11] state that the next logical step should involve systems that aid rather than replace human decision makers and have called these "expert support systems."

Simon observed that "When we ask how executives in organizations make nonprogrammed decisions, we are generally told that they 'exercise judgment,' and that this judgment depends, in some undefined way, upon experience, insight, and intuition." Simon took issue with such an explanation when he remarked, "To name a phenomenon is not to explain it. Saying that nonprogrammed decisions are made by exercising judgment names that phenomenon but does not explain it" [8].

In contrast, the AHP makes a significant contribution toward understanding this important phenomenon of complex, nonprogrammed decisions. By allowing decision makers to model a complex problem into a structure that shows the relationships of goals, criteria, uncertainties, and alternatives, it allows for the application of experience, insight, and intuition in a logical and thorough way.

REFERENCES

1. R. L. Cook and K. R. Hammond, "Interpersonal Learning and Interpersonal Conflict Reduction in Decision-Making Groups," in *Improving Group Decision Making in Organizations: Approaches from Theory and Research*, Academic Press, New York, 1982.
2. Lyn Jones, *Decision Analysis*, The Open University Press, Milton Keynes, Great Britain, 1975.
3. C. H. Kepner and B. B. Tregoe, *The Rational Manager: A Systematic Approach to Problem Solving and Decision Making*, McGraw-Hill, New York, 1965.
4. James Martin, *Design of Man-Computer Dialogues*, Prentice-Hall, Englewood Cliffs, NJ, 1973.
5. G. A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Information Processing," *Psychol. Rev.*, 63(2), 81-97 (March 1956).
6. Thomas L. Saaty, *Decision Making for Leaders*, Lifetime Learning Publications, division of Wadsworth, Inc., Belmont, CA, 1982.
7. L. L. Whyte, *Hierarchical Structures*, American Elsevier, New York, 1969.

8. Herbert A. Simon, *The New Science of Management Decision*, Harper and Brothers, New York, 1960, pp. 40-43.
9. Thomas L. Saaty, *The Analytic Hierarchy Process*, McGraw-Hill, New York, 1980.
10. Ernest H. Forman, Thomas L. Saaty, Mary Ann Selly, and Rozann Waldron, *Expert Choice*, Decision Support Software, McLean, VA, 1983.
11. F. L. Luconi, T. W. Malone, and M. S. Scott-Morton, "Expert Systems and Expert Support Systems: The Next Challenge for Management," Sloan WP No. 1630-85, Center for Information Systems Research, Massachusetts Institute of Technology, Cambridge, MA, 1985.

ERNEST H. FORMAN

MARY ANN SELLY

ANIMATION OF NETWORK-BASED SIMULATION MODELS

INTRODUCTION

Computer simulation has been one of the most widely used operations research tools over the past 25 years. One reason for this is its broad range of applicability to difficult systems design and operation problems. The past 15 years have seen the development of many software packages to perform the usual simulation functions of timekeeping, statistical data collection, model definition, and output reporting. Some of these modeling tools view systems as sets of interrelated activities that can be represented as process flow diagrams or networks of nodes and arcs. This process view has many advantages in terms of model conceptualization, computerization, and communication with nontechnical audiences. The software permits systems description via input rather than requiring the modeler to develop FORTRAN-like subroutines to handle the logic of system state changes. Some of the better known network-based languages include GPSS, Q-GERT, SLAM, GEMS, and SIMAN (1-3). These packages typically provide the modeler with a numerical and graphical analysis of the system behavior over some period of time, that is, systems span times, resource utilizations, queue lengths, and so forth. These statistics are very meaningful to the experienced modeler and can be of considerable value in predicting system performance.

A common problem in practice, however, has been that these results are often not particularly meaningful to nontechnical audiences, especially those levels of upper management that usually make the commitments to implement systems concepts into action. Recognizing the problem, attempts have been made to make the model input and output more self documenting, to provide better statistical graphics, and to create animations of model activity during the course of the simulation.

Considerable effort has gone into animation and visual scene generation in recent years, often requiring considerable resources in computing power, time, and effort to set up the animation. Certain military applications (e.g., flight simulators) make use of extremely complex graphics capabilities. On a less complex scale, several packages have been developed for animation of activity in discrete event and process/network-type models. The advent of the microcomputer and its graphics capabilities have provided the means for low-cost portable animations. Much animation work has been done using the microcomputer, although little has been written in the literature by nonvendor users concerning methodologies and observations based on actual practice.

The purpose of this article is to provide some user insight into animation methodologies and experience in actual practice. A brief survey of animation software is provided in order to educate the reader as to the type of available tools. A microcomputer-based approach to animation is then out-

lined in detail, as well as its application in the study of a flexible machining cell. Several observations from this case study are presented, some of which apply to the use of animation in general.

REVIEW OF GRAPHICS ANIMATION PRODUCTS

Pope (4) gives a sample of graphics animation packages, as shown in Table 1, with an emphasis on manufacturing applications. These animation products generally fall into two categories. The first group comprises highly detailed models of the geometry and logical control of small sets of equipment and work pieces. Products such as PLACE and ROBOGRAPHIX are primarily directed at modeling robotics cells. These systems include libraries of three-dimensional (3d) geometric models of robots, and the user may define additional robotic components, work pieces, fixtures, and so forth. NCVISION is representative of products that generate NC cutter path instructions from a geometric definition of work piece, fixtures, and cut geometry. The software produces a 3d animation of the spinning cutter going along the cutter path, thus reducing the amount of NC proofing with cuts in actual material.

The second group of products comprises high-level, general application simulations that focus on the flow of items through the system and the status of queues and activities through time. The Real Time Conveyor Simulator RTCS 250 is unique in that it interfaces directly with programmable controller logic for operating conveyor systems. The color screen shows two-dimensional (2d) animations of items flowing through the conveyor and the status of sensors, switches, and so forth. The physical conveyor layout and other systems parameters are defined by tables in a host computer and passed to the RTCS. The application of RTCS in a flexible manufacturing systems study is discussed by Hitchens (5).

One of the most extensive general-purpose animation packages is AutoMod/AutoGram. AutoMod is a GPSS (General Purpose Simulation System)-based simulation package that includes libraries of GPSS logic for manufacturing equipment and control strategies. The user can modify and extend the GPSS logic as needed. If animation is desired, an event trace file is generated, which records all systems events as they occur through time. The manufacturing components in AutoMod have corresponding 3d geometric models defined in AutoGram. The 3d animation of the event trace is generated on CAD-type graphics equipment. A case study involving a material storage and retrieval system is given by Stauffer (6).

SEE WHY provides the analyst with a collection of FORTRAN subroutines to perform various simulation and graphics functions. The user builds his/her own simulation using these subroutines, and the 2d animation can be structured as desired. One version is available on a microcomputer, resulting in a highly portable set of equipment for demonstrating animation to management audiences.

TESS (The Extended Simulation System) is an extension of the widely used SLAM (Simulation Language for Alternative Modeling) simulation software, incorporating the results of research on data bases, decision support, and graphics. There is a capability for generating a 2d animation of the status of queues and activities as they change through time. The capabilities of TESS are the outgrowth of several years of animation work, including various unique applications such as animating security and medical actions at football games.

SIMAN is a general-purpose simulation language with certain features for modeling material-handling devices that make this tool especially suitable for manufacturing applications. The PLAYBACK module will show up to six dynamically changing bar charts for queue lengths and resource utilizations. The CINEMA module provides a more general real-time animation capability for SIMAN models.

MAST (Manufacturing Systems Design Tool) is specially designed for simulation of the hardware and control logic of flexible manufacturing systems (FMS). Options include many possible material-handling and control logic modules. An event trace is downloaded to microcomputer, which performs a 2d animation of the FMS.

These procedures vary considerably in their level of detail and scope of application. The cost of the software (and hardware, in some cases) varies from about \$500 to over \$100,000. Some packages offer a menu-driven, no-programming model and animation definition capability for the nonsimulation user. Other packages interface with network-type or FORTRAN simulation models to provide a more flexible animation tool for the experienced simulation modeler. The appropriate choice of animation tools depends on the user's need, potential cost benefit, quality of input data, available modeling skills, and other factors.

A MICROCOMPUTER-BASED APPROACH

After evaluating the available graphics animation packages, it was felt that a less expensive, more generally applicable technique was needed. In order to meet these objectives, the scheme illustrated in Figure 1 was developed. The procedure involves the following steps:

1. Develop a network-style simulation model of the system being analyzed. In this particular case, an in-house-developed modeling language, SYSMODL (Systems Modeling Language), was used, but any of the popular languages such as SLAM, GEMS, GPSS, or SIMAN could be used.

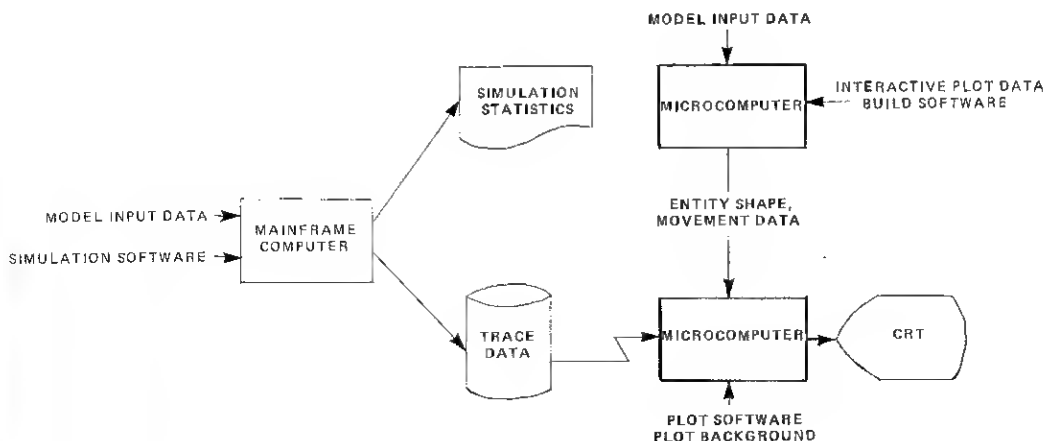


FIGURE 1 Approach to graphics animation.

TABLE 1 A Sample of Graphics Animation Packages

Product	Vendor/Developer	Equipment	Notes
Positional Layout and Cell Evaluator (PLACE)	McDonnell Douglas Automation Co. Box 516 St. Louis, MO 63166	VAX, R-100 graphics station	3d representation of standard robots and work pieces. Interfaces with modules for programmed robot instructions, error detection, etc.
NCVISION, ROBOGRAPHIX	ComputerVision 201 Burlington Rd. Rt. 62 Bedford, MA 01730	CV graphics station	NCVISION generates NC code from work piece and cut geometry and performs 3d animation of cutter path. ROBOGRAPHIX is for modeling of robotic applications.
ROBOCAM	SILMA, Inc. 2111 Grant Road Los Altos, CA 94022	Apollo graphics station	3d representation of standard robots, user-defined components, etc. Allows programming and simulation of motion.
Interactive Robot Programming and Simulation System (IRPASS)	Nokia Corp. Finland	Calma 1000 Graphics System, LSI 11/23 robot processor	Enables programming and simulation of all robot commands; robot independent.
ROBOT-SIM, ROBOT-PRO	General Electric Calma	Apollo or VAX 700 series, Calma graphics station	Programming and simulation of robots and other work cell components. Libraries of standard robots.
Real Time Conveyor Simulator RTCS 250	HEI Corp. 455 East Kehoe Blvd. Carol Stream, IL 60187	RTCS includes color display, console, and PC interface. Host computer required.	Performs 2d animation of conveyor operation while interfaced directly to programmable controller logic.

AutoMod, AutoGram	AutoSimulations, Inc. P.O. Box 633 562 South Main St. Bountiful, UT 84010	Silicon Graphics Iris Workstation, VAX 750 or IBM mainframe 1 meg memory, 10 meg disk	AutoMod performs simulation using GPSS modules for equipment and control. AutoGram allows 3d anima- tion of equipment and entity move- ment.
SEE WHY	Istel, Inc. 83 Cambridge St. Burlington, MA 01803	Mainframe or Cromemco Z-2H micro, graphics on Intelligent Systems 8051	Collection of FORTRAN subroutines for constructing simulation models and real-time 2d animation.
The Extended Simulation Systems (TESS)	Pritsker & Assoc. P.O. Box 2413 W. Lafayette, IN 47906	Mainframe, graphics terminal	TESS includes the SLAM simulation language and allows animation of activity and queue status.
SIMAN (PLAYBACK and CINEMA modules)	Systems Modeling Corp. P.O. Box 10074 State College, PA 16805	SIMAN/PLAYBACK runs on PC. CINEMA runs on PC/AT with graphics board and hi res screen	SIMAN has general simulation capability. PLAYBACK is dynamic bar graphs for queues and resources. CINEMA is real-time animation of SIMAN simulation on user-defined 2d ani- mation.
Manufacturing Systems Design Tool (MAST)	CMS Research, Inc. 945 Bavarian Ct. Oshkosh, WI 54901	Mainframe, graphics on micro	General flexible manufacturing system simulator; uses micro for 2d anima- tion.
FMSim	Robotics Technology One Kingwood Place Kingwood, TX 77339	RT System One 40 meg hard disk Seiko GR-1104 terminal, GR-3100 Data Tablet	Menu-driven model construction for 2d animation. Little knowledge of simu- lation required. Produces real-time simulation and statistics.

(Continued)

TABLE 1 (Continued)

Product	Vendor/Developer	Equipment	Notes
Interactive Simulation with Graphics Tools (INSIGHT)	Battelle Pacific Northwest Laboratories P.O. Box 999 Richland, WA 99352	Mainframe computer, high res scope, digitizer pad	Interactive graphical preprocessor, run time animation routines, and postprocessor for SLAM models.
PCModel	Simulation Software Systems 2470 Lone Oak Dr. San Jose, CA 95121	IBM PC, 128K	80 x 24-character grid displayed as window on larger logical grid as defined by user. Directed at animation of assemblies moving through manufacturing operations (2d).
ModelMaster	General Electric Automation Controls Div. Charlottesville, VA 22906	PC, 612K, or VAX 11/780 with Lexidata 3400 or Tektronix 4115 screen	Menu-driven, no-programming user definition of simulation parameters (2d).

2. Execute the model and generate an event trace file, sorted by the time of event occurrence. Most simulation languages have a trace capability, although the structure of the trace varies. In this case, the trace included records of four types;
 - Type 1. Activity start event — Activity number, start time, end time (where activity means a time-consuming operation in the model logic).
 - Type 2. Activity and event — Activity number, activity end time.
 - Type 3. Queue change event — Queue number, time of change, amount of change in queue level.
 - Type 4. Resource change event — Resource number, time of change, new resource level.
3. Download the trace file to a microcomputer. The alternative here is to execute the model directly on the microcomputer. The trade-off is the download time versus the model execution on the microcomputer, which may be excessively slow. Both approaches have been used successfully in the test case discussed below.
4. Define the plot background using a screen drawing package (several inexpensive packages are available). Save this screen image in a binary file that can be called by the animation program. It has been found in our experience that backgrounds that visually match the application are preferable to backgrounds that reflect the simulation network logic.
5. Define the shapes corresponding to the entities that will move across the screen in activities, reside in queues, and represent units of resources.
6. Define the screen plot locations and movements corresponding to each activity, queue, and resource in the model. This process can be tedious but is aided in this case by an interactive program that scans the model input data and queries the user for the necessary plot information in any set of coordinates the user wishes to use. This interactive program is a key to reducing the time and effort in creating animations of models. Movement paths are assumed to be piecewise linear. Numeric counters for activity and queue levels are placed at the desired location on the screen if needed.
7. Execute the animation program (GRAFSIM), which steps through time in increments and updates the screen image based on events in the trace file. GRAFSIM interpolates over the entity movement paths to locate the current position of the entity shapes. A key to plotting speed is the ability to draw and erase shapes along the path without having to regenerate the effected areas of background. This was accomplished in this case by using the appropriate BASIC graphics commands. The time increment can be interactively increased or decreased to speed or slow the animation process. The animation procedure is outlined in the animation algorithm below.

Animation Algorithm

1. Set the clock time $T = 0$ and the time step DT to the desired value. Read the first event from the trace file. The time of occurrence of this event will be denoted ET .

2. Compare the current clock time T to ET . IF $T < ET$, go to step 3; otherwise, go to step 4.
3. Update the plot locations of the shapes for the activities in the in-process list. Increase or decrease the time step DT based on interactive input from the keyboard. Increment the clock, $T = T + DT$. Go to step 2.
4. Process this event at time ET . If there is no corresponding plot data, proceed to step 5. Otherwise, process according to the type of event:
 - Type 1. Activity start — Add activity to in-process list.
 - Type 2. Activity end — Remove activity from in-process list.
 - Type 3. Queue level change — Update plot of items in queue.
 - Type 4. Resource level change — Update plot of units of resource.
5. Read next event from the trace file and set ET to its time of occurrence. Go to step 2.

TEST CASE—A FLEXIBLE MACHINING CELL SIMULATION

As a test case, a flexible machining cell (FMC) was modeled using the approach described above. The objectives were to measure the effort involved in data entry to define the animation parameters and to measure the informational benefits resulting from the animation. The subject of the animation was the FMC at the LTV Aerospace and Defense Company—Vought AeroProducts Division. This FMC has eight horizontal four-axis machining centers, two coordinate measuring machines, a wash station, two automatic pallet changers, and four automatic guided vehicles (AGVs). The FMC performs machining for 541 different parts for the B-1B bomber program at a maximum rate of 14 pieces per hour. Parts are fixtured to one of seven types of tooling "risers" attached to a pallet base. Each part is fixtured and run through the cell and refixed as many times as necessary to make all cuts. The uniqueness of this cell lies in the large number of different part types handled at relatively low demand rates per part.

The cell was simulated in the GEMS, SLAM, and SYSMODL languages under a variety of conditions, and the usual statistical results were generated. The simulation study had a direct impact on the decisions concerning numbers of automatic work changers (\$100,000 each), AGVs (\$60,000 each), pallet/risers of each type (\$10,000 each), and inspection machines (\$900,000 each).

The current SYSMODL model contains 127 activities, 62 queues, and 47 resources, in addition to other logic for reading work orders, routing the AGVs, and so forth. An animation of this FMC model was created using the approach outlined in the previous section. Figure 2 shows screen prints of four instants in the animation. The experience with animating the FMC brought about the following observations:

1. The value of the animation was almost solely in conceptual presentation rather than in hard analysis. (This does not mean that the animation serves no purpose, seeing that these activities are a critical part of any implementation project.)

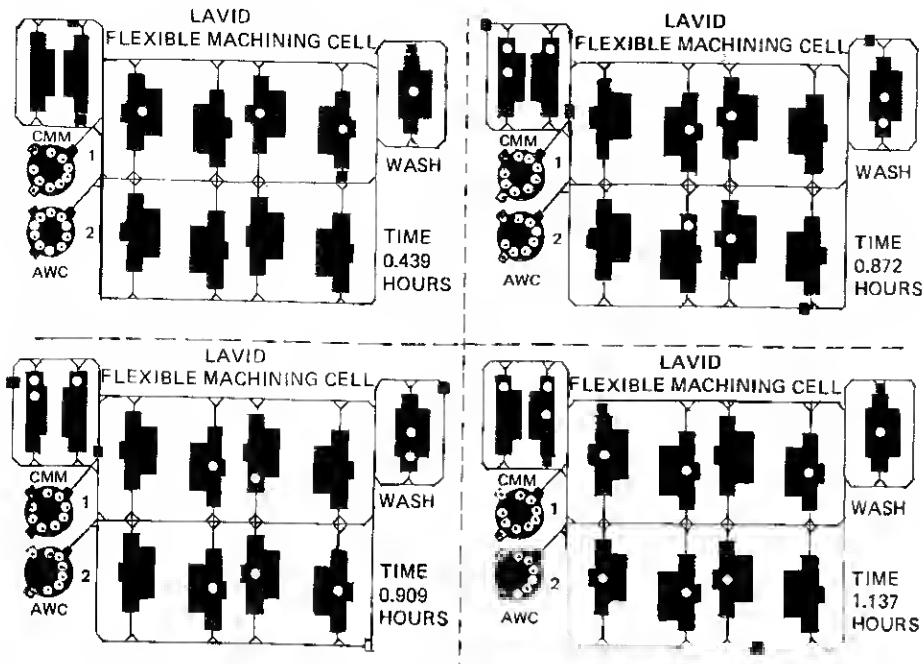


FIGURE 2 Snapshots of flexible machining cell animation.

2. The time frame of the animation (in this case, 4 hours of FMC operation) was too short for any concrete insights. It was necessary to simulate the FMC for 10,000 hours of scheduled operation in order to answer certain questions, and it would be impracticable to animate the system long enough to obtain this type of information.
3. Animation must have a windowing capability to view larger systems than the FMC (see Fig. 2). Color is preferable to monochrome for presentations, even with the sacrifice in resolution. The resolution of the screen limits the amount of text and numeric information that can be displayed without creating clutter.
4. The time for the microcomputer to generate the animation is a function of the amount of logic being performed in the program; thus, the animation viewing time is not necessarily proportional to the time frame simulated. Experiments showed that for the intended use, the proportionality was fairly constant and was not a major issue. If animation time must be forced to better track simulated time, compensating logic can be incorporated into the software.
5. The microcomputer was able to produce animations acceptably fast for viewing while maintaining a fluidness of motion of the entities moving through the system. (About 30 items were in process at one time.) Compiled BASIC was found to produce animations about six times faster than interpreted BASIC, thus producing much superior results. It is believed that FORTRAN would produce even better results if appropriate graphics subroutines for microcomputers were available.
6. The data entry time for definition of the background and movements of the items was low enough to be attractive from a time/cost

standpoint. The data entry time for the FMC animation background and movement data for each portion of the model was approximately 4 days.

7. The approach of linking the simulation and animation via a trace data set has a potential problem in that the trace data file can quickly become very large. Four hours of FMC operation required 200Kb of disk space. The alternative is animate "on the fly" as the system is simulated, but the simulation may be unacceptably slow and the timing problem alluded to in observation 5 is aggravated. This occurs because the animation time reflects variances in the amount of simulation logic in addition to variances in the amount of screen plot updating.
8. Large screen projectors were found to be an effective means of presenting microcomputer-based graphics to large audiences. In fact, few other acceptable means were found for reproducing or enlarging the microcomputer screen image.
9. Some processes within the system will likely require some special plotting procedures. In this case, the items (pallets) around the oval-shaped work changers required special logic to plot the items appropriately.
10. It may be necessary to model the system in more detail than usual in order to obtain realistic animation. Deviations from real life behavior are quickly noticed by nontechnical audiences, and attention is drawn from important points. In the FMC animation, for example, AGVs will occasionally collide. This logic was not handled in the simulation because the low density of AGVs in the system makes this event inconsequential in terms of system performance, yet its occurrence in the animation creates a distraction for some viewers.

Several extensions to the methodology outlined above are possible. The first enhancement would be to handle 3d shapes. The trade-off would be computational speed (and, thus, fluidness of animation), ease of model plot data digitizing, and screen clutter. Another enhancement would be to define movement of activity shapes relative to the current location of other activity shapes. In this manner, it would be possible to model the motions of a multi-axis robot arm, for example, as a network-type model. The movements of fixed elements of the arm are activities in the model, and their animation movements are defined relative to each other, so that the shapes for the various components move in a connected manner.

CONCLUSIONS

Advances in microcomputer computational speed and graphics capabilities, as well as their decreasing cost, make them ideal for generating animated simulations. In order to reduce animation development time and costs, network-based simulation models can be easily digitized relative to an appropriate background. This approach offers a quick and inexpensive way of presenting systems concepts and simulation results to nontechnical audiences. Continued development in microcomputer-based graphics and simulation software will open the way to many new applications of animated simulation in the future.

REFERENCES

1. "Catalog of Simulation Software," *Simulation*, 41(4) (October 1983);
ibid 42(2) (February 1984).
2. A. A. B. Pritsker and C. D. Pegden, *Introduction to Simulation and SLAM*, Wiley and Sons, New York, 1979.
3. R. E. Shannon and D. T. Phillips, "Comparison of Modeling Languages for Simulation of Automated Manufacturing Systems," in *AUTOFACT 5 Conference Proceedings*, Detroit, MI, November 1983.
4. D. N. Pope, "A Review of Graphics Animation of Manufacturing Systems," in *AUTOFACT 6 Conference Proceedings*, Anaheim, CA, October 1984.
5. M. W. Hitchens, "Simulation: The Key to Automation Without Risk," CAD/CAM Technology, Society of Manufacturing Engineers, Fall 1984.
6. R. N. Stauffer, "Graphic Simulation Answers Preproduction Questions," CAD/CAM Technology, Society of Manufacturing Engineers, Fall 1984.

DON N. POPE
M. C. STROUD

ANSI—THE AMERICAN NATIONAL STANDARDS INSTITUTE

INTRODUCTION

The American National Standards Institute (ANSI) is a federation of standards-using and standards-developing organizations responsible for coordinating and promoting the voluntary development of national standards by concerned parties in the United States. This nongovernmental body works closely with the private sector and many government agencies to encourage participation in voluntary standardization activities and the use of voluntary standards whenever possible.

ANSI also manages and coordinates U.S. participation in international standardization efforts as the U.S. representative to ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission). It maintains contact with other standards-making bodies, such as CEN (European Committee for Standardization), CENELEC (European Committee for Electrotechnical Standardization), Consultative Committee on Telegraph and Telephone (CCITT) of the International Telecommunications Union (ITU), and signatories of the General Agreement on Tariffs and Trade (GATT).

As a major publisher and distributor of standards, ANSI accommodates other standards-developing bodies who publish their own standards and make copies available through ANSI. It also acts as an information clearinghouse by providing informative documentation on both national and international standards and standards-making bodies.

HISTORY AND STRUCTURE

ANSI has had numerous names since it was first established in 1918 as the American Engineering Standards Committee by the American Society for Testing of Materials (ASTM), American Society of Mechanical Engineers (ASME), the American Society of Mining and Metallurgical Engineers, the American Society of Civil Engineers, and the American Institute of Electrical Engineers. In 1928, it was reorganized as the American Standards Association and became incorporated in 1948. In 1966, it then became the United States of America Standards Institute, and in 1969, finally became the American National Standards Institute.

ANSI assists organizations involved in standardization to reach agreement cooperatively on needs for standards, establishing priorities, plans to meet identified needs, and avoiding duplication of effort. ANSI offers U.S. standards-developing organizations a neutral forum for resolving differences and provides procedures and services to help them use their resources effectively. ANSI coordinates the development of standards for everything from a specific film of varying speeds for cameras, screwthread configurations for nuts and bolts, and local area networks for computer interconnection, to other standards related to health and safety factors, design considerations, and performance specifications.

ANSI's members come from approximately 1,000 large and small companies, 235 trade, technical, professional, labor, and consumer organizations, and 20 government agencies. Table 1 shows a small sample of the types of groups who have membership in ANSI.

ANSI's headquarters are in New York, operating with staff of 100 professional and support personnel. The staff includes a president, two vice presidents, and staff directors for its principal functions. ANSI also maintains Washington offices for government liaison functions. The organizational charts (Tables 2 and 3) show ANSI's organizational components and the hierarchy of its supporting staff.

STANDARDS DEVELOPMENT

By providing guidelines for standards-making procedures, ANSI ensures that consensus is reached by those affected by the standards. Standards must be developed using an open process for comment and review, thereby reconciling major objections by those concerned. Although standards do not require unanimous support, this process allows all points of view to be considered and dealt with fairly. Evidence that consensus exists is provided by organizations and committees that voluntarily submit standards to the institute for approval.

When it has been shown that all substantially affected interests have been given an opportunity to participate in the development of a standard or to comment on its provisions, ANSI conducts an independent verification of the adequacy of consensus and approves the standards as American National Standards.

Standards are brought to ANSI via three different methods: accreditation, canvassing, or standards committee methods.

Accreditation may be requested from ANSI's Certification Committee by any organization involved in, or desiring to be involved in, standards work. Once accredited, the organization may sponsor standards using a development of consensus procedure comparable to that required by the standards committee method. The sponsor takes on the responsibility of developing, publishing, and getting consensus on a standard. After the proposed standard has been approved within an accredited organization, it is submitted, with all documentation showing consensus, to ANSI to be approved as an American National Standard. ANSI then verifies that consensus has been reached. One such accredited organization is the EIA, which has developed the most widely used standards for computer equipment in North America.

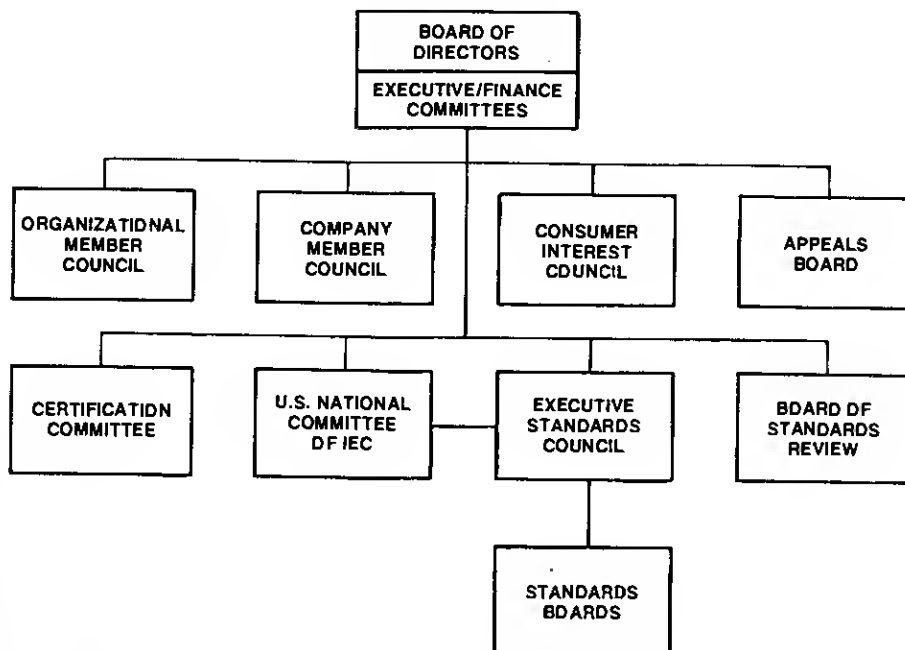
Canvassing is usually done by an organization with an existing or draft standard that it wants to have approved by ANSI. The organization takes a canvass or mail poll of all organizations that are known to have interest and concern in the subject, thereby becoming the sponsor of the standard. Not only is the sponsor responsible for preparing the poll but it is also responsible for submitting all pertinent documentation to ANSI for further processing. The documentation must include the standard being proposed, the canvass list, the comments received, and any responses by the sponsor to adverse comments received. Standards are often revised after the initial canvass to accommodate negative responses.

Establishment of a standards committee may be requested when organizations have developed standards on the same or related subjects or when a party thinks a standard should be developed for a particular subject. The

TABLE 1 Selected ANSI Participating Members

Organizations	Government Agencies	Companies
American Society of Mechanical Engineers (ASME)	National Bureau of Standards	AT&T
Institute of Electrical and Electronics Engineers (IEEE)	U.S. Department of Defense	Data General Corp.
Computer and Business Equipment Manufacturers Assn. (CBEMA)	U.S. Department of Energy	Exxon Corp.
American Society for Testing of Materials (ASTM)	U.S. Department of Agriculture Rural Electrification Admin.	International Business Machines (IBM)
Electronic Industries Assn. (EIA)	General Services Admin.	OCLC, Inc.
Underwriters Laboratories (UL)	U.S. Department of the Interior	Rabbit Software Corp.
Toy Manufacturers of America	City of Shreveport, Louisiana	Microbot, Inc.
Optical Laboratories Assn.		B.H.&P., Inc.
The Irrigation Association	New York Power Authority	TRW, Inc.
		Trus Joist Corp.

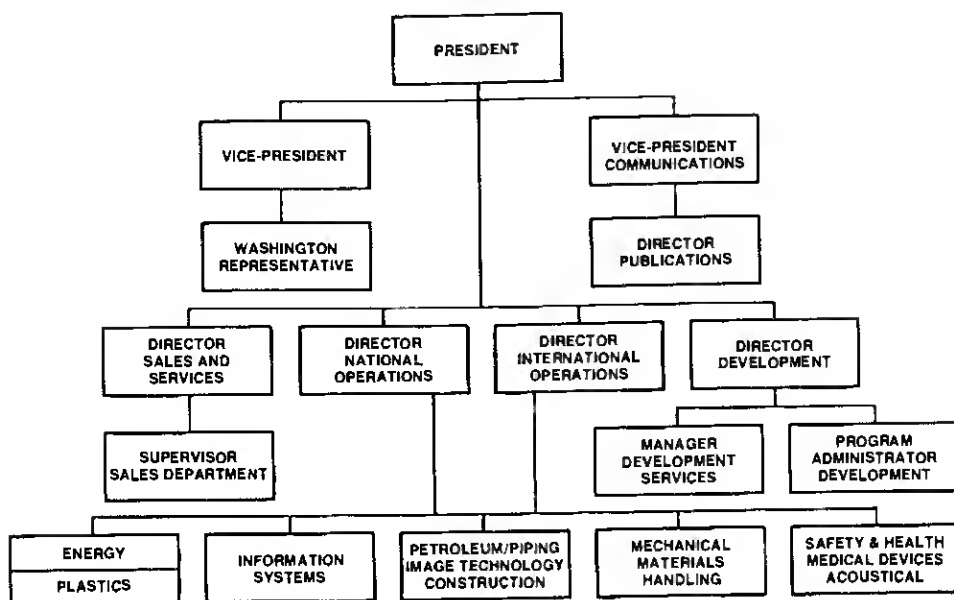
TABLE 2 Organization Chart



Board of Directors:	The governing and policymaking body of the Institute.
Executive/Finance Committees:	Executive Committee provides guidance for administration of ANSI and acts for the Board of Directors between meetings. Finance Committee maintains a continuing review of ANSI financial affairs and makes recommendations to the Board.
Organizational Member Council and Company Member Council:	Ensure participation of their members in ANSI programs and provide a communication channel between their members and constituents and ANSI's Board on programs and policies of the Institute; help determine standards needs.
Consumer Interest Council:	Responsible for consumer input to standards programs; reviews all consumer standards; represents and protects consumer interests in national standards activities.
Appeals Board:	Hears complaints of those who believe they have been adversely affected by the action or inaction of any ANSI board or council.
Certification Committee:	Develops and operates programs leading to national accreditation of certification programs.
U.S. National Committee of IEC:	Responsible for effective participation in the work of IEC and for operation of the technical advisory groups that develop the U.S. position on international electrotechnical standards.
Executive Standards Council:	Manages the standardization activities coordinated by ANSI—promulgates operating procedures; stimulates expeditious completion of standards work; also coordinates U.S. participation in technical work of international organizations.
Board of Standards Review:	Approves standards as American National Standards and acts on withdrawal and reaffirmation when it finds that a consensus exists among those substantially concerned with the scope and provisions of the standards under consideration.
Standards Boards:	Assist the Executive Standards Council in carrying out its management and coordination functions for standards development in the discipline or homogeneous technical sphere in which the particular Standards Board operates.

request is forwarded to the Executive Standards Council, which assigns the subject to a Standards Management Board which, in turn, appoints a secretariat. A poll may be taken by ANSI to determine whether there is enough interest to establish a standards committee. If there is enough interest to warrant a standards committee, notices are issued to the press to this effect and interested parties are encouraged to participate.

TABLE 3 Staff Organization Chart



For example, ANSI assigned a secretariat, CBEMA, the responsibility of forming the corresponding ANSI Committees X3 and X4, which have merged to form Committee X3 (Information Processing Systems). X3 now has responsibility for all of ANSI's computer-related standards. The standards committee is comprised of a secretariat (in this case, CBEMA) and a balanced representation of consumers, producers, and general interests. ANSI monitors the progress of the standards committee and acts on its output.

The secretariat organizes and appoints officers to the standards committee and coordinates the administrative functions. These administrative functions include

1. Ensuring that all required documentation has been submitted and distributed to committee members for review prior to taking a formal ballot.
2. Taking a formal ballot, the results of which remain confidential to the secretariat and the committee officers until the ballot period closes.
3. Ensuring that changes to the balloted draft are made to resolve negative responses, at which time another ballot is submitted to the committee and all committee members must reaffirm their vote or cast a negative vote.
4. After the balloting process and consensus, the proposed standard must be published in its entirety, or a notice is published indicating that the standard is available.
5. The secretariat uses its discretion as to whether or not the proposed standard is ready to be submitted to the ANSI Board of Standards Review for approval as an American National Standard. If at least two thirds of the committee members voting have approved the

standard, it is mandatory that the proposed standard, with its documentation, be submitted to ANSI.

When the proposed standard reaches ANSI, it is examined by the staff to determine that the documentation required has been forwarded and that evidence of consensus exists, just as was done when the proposed standard was submitted to the secretariat for a ballot by the standards committee. The proposed standard is now submitted to the vote of the Board of Standards Review, which requires an affirmative vote of not less than two thirds of the full board, taken by written ballot.

The standards developer may assign its numeric or alphanumeric designation (i.e., ASTM E731-80, Guidelines for the Procurement of Commercially Available Computerized Systems). If no designation is assigned by the standards developer, ANSI assigns one using the standards-developing organizations acronym (i.e., ABC 123-1986). If the standard has been developed by an accredited standards committee, the secretariat assigns the designation using the committee's alphanumeric identification (i.e., X3.95-1982, Hexadecimal Input/Output to Microprocessors Using 5-Bit and 7-Bit Teleprinters). Upon ANSI approval, the designation is prefixed by "ANSI" (i.e., ANSI X3.95-1982, ANSI/ASTM E731-80, etc.).

INTERNATIONAL PARTICIPATION

The international standards-making process for ISO and IEC is comparable to ANSI's process. Nearly a quarter of ANSI's annual budget is spent on administration of international standards activities and dues to ISO and IEC. ANSI's funding comes from membership dues and the sale of its publications.

ISO was established to achieve international agreement on industrial and commercial standards. With international agreement on standards, international trade and the interchange of scientific and technological data were aided and expanded. ISO membership consists of representatives from national organizations and other concerned groups. Of ISO's more than 2,400 technical committees and subgroups, ANSI holds the secretariats of approximately 285. ANSI participates in most of ISO's technical work and has observer status in the areas in which it is not an active participant. The ISO Technical Committee 97 (Computer and Information Processing) and Technical Committee 95 (Office Machines) have merged together to form Technical Committee 97 (Information Processing Systems).

IEC was the first international standards organization that was concerned with coordinating and unifying national standards for international use. IEC and ISO develop standards through international consensus processes. The U.S. National Committee, which ANSI administers, is involved in IEC's entire technical program. Of IEC's more than 180 technical committees and subcommittees, ANSI holds the secretariat of 13 technical committees and 20 subcommittees.

In 1985, ISO and IEC joined forces to provide standards in the information technology sector, covering microelectronics, telecommunications, and computing and information processing. The primary objective of this cooperative venture is to develop international standards to facilitate the communication and interconnection of equipment, such as personal computers, word processors, and other industrial, commercial, and domestic systems.

TABLE 4 Standards-Making Bodies^a

Organization	Types of Standards	Examples
American National Standards Institute (ANSI)	A wide variety, covering data formats, coding, safety, software, test procedures, definitions. Some are jointly published with other organizations.	National Electrical Code, ASCII, FORTRAN, flowchart symbols.
American Society for Testing of Materials (ASTM)	Standards tests for strength and performance of materials and substances; procedures for calibration of testing equipment.	Temperature, heat, strength of metals in certain applications, strength of bonds and connections.
International Telegraph and Telephone Consultative Committee (CCITT)	Standards related to telecommunications.	Modems and interfaces, protocols, formats.
Canadian Standards Association (CSA)	Primarily safety-related standards.	CSA 566B for electrical and electronic test and measuring equipment.
Deutsches Institut für Normung (DIN)	Standards covering form factors and ergonometrics, among other things.	Panel meter cutouts, keyboard and CRT dimensioning, printed circuit card size and connectors.
Electronics Industries Association (EIA)	Interface and signaling standards.	RS-232, RS-422 for communications; RS-170 for base-band video.
Federal Communications Commission (FCC)	emi/rfi standards.	Docket 20780, Class A and Class B.

Federal Information Processing Standards (FIPS)	Standards covering data codes, signaling, data encryption, parity.	PUB 46: Data Encryption Standard.
International Electrotechnical Committee (IEC)	Standards covering safety, procedures, definitions.	IEC 225: octave, half octave, and third octave band filters.
Institute of Electrical and Electronics Engineering (IEEE)	Extremely wide variety, covering test procedures, definitions, interface standards, safety, networks.	IEEE 488 interface bus, 472 surge withstand, 770 Pascal language, 802 networks.
Instrument Society of America (ISA)	Standards for installation, calibration, dimensions, and definitions of process-control transducers and instrumentation.	S50.1 Compatibility of Analog Signals for Electronic Industrial Process Instruments.
International Organization for Standardization (ISO)	Standards for measurement, data encoding, signal transmission.	ISO 646.7 Bit Character Set for Information Processing Interchange.
National Bureau of Standards (NBS)	Fundamental physical measurements and calibration.	NBS 125 Thermocouples; Data Encryption Standard 1977.
Underwriters Laboratories (UL)	Safety and related performance of equipment and components.	UL 1244 Electrical and Electronic Measuring and Testing Equipment; UL 94 Flammability of Plastic Materials.
Verband Deutscher Elektrotechniker (VDE)	Standards for safety, form factors, performance, emi/rfi.	VDE-0871A for emi/rfi.

^aReprinted with permission from the July 1984 issue of *Computer Design*, © 1984, PennWell Publishing Company, Advanced Technology Group.

TABLE 5 Selected Standards Affecting Microcomputers

Project/Standard Designation	Subject or Title	Stat ^a	Act ^b	Meth ^c	Comments, Expected Date of Transmittal to ANSI, Etc.
IEEE 802.1	Local Network for Computer Interconnection	NS	2	O	
IEEE 802.3 1985	Local Area Network Standard — CSMA/CO Access Method and Physical Layer Specifications	AP	0	O	Current
IEEE 802.3a 1986	LAN CSMA/CD, Sect. 10: Medium Attachment Unit and Base-Band Medium Specifications for Type 10 Base 2	AP	0	O	Current; supplement to ANSI/IEEE 802.3-1985
IEEE 942	Digital Semiconductor Test Programming Language	NS	2	O	
EIA RS-404 1978	Start-Stop Signal Quality between Data Terminal Equipment and Nonsynchronous Data Equipment	RV	5	O	1985; EIA/TR-30.1 Committee Pub rev 2/1/85-4/2/85
EIA RS-496 1984	Interface between Data Circuit-Terminating Equipment (DCE) and the Public Switched Telephone Network	AP	0	O	Current; EIA/TR-30 Committee
EIA RS-511	User Level Format and Protocol for B1- Direct Transmission of Information between a DNC System and Numerically Controlled Machines	NS	2	S	EIA/IE-31 Committee (SP 1393) Assign: ISSB/EESB
EIA 334A 1985	Signal Quality at Interface between DTE and Synchronous DCE for Serial Data Transmission	AP	0	O	Current; EIA/TR-30.1 Committee Revision of EIA RS-334-1968

X3.140 1986	Information Processing Systems-OSI- Connection-Oriented Transport Layer Protocol Specification	AP	0	S	Current; Project 332-M, X353 Corresponds to ISO/DIS 8072 and 8073
X3.130	Intelligent Peripheral Interface, Device- Specific Command Set for Disk Drives	NS	5	S	1986; Project 467-D, X3T9 2nd Pub rev 12/31/85-2/6/86
X3.131	Small Computer Systems Interface (SCSI)	NS	5	S	1986; Project 375-D, X3T9.2 2nd Pub rev 1/3/86-3/4/86
X3.134.1	8-Bit ASCII Structure and Rules	NS	5	S	1986; Project 495-D, X3L2 Pub rev 3/15/85-7/14/85
Z39 SC BB	Standard Computer Software Number (SCSN)	NS	5	S	1986
Z39 SC EE	Compact Disc Data Format	NS	2	S	
Z39 SC G	Common Command Language for Use in Interactive Information Retrieval	NS	2	S	1986; development within ISO/TC46/SC4/WG5
Z39.39 1979	Compiling Newspaper and Periodical Publishing Statistics	RF	4	S	1986; Pub rev 2/28/86-4/27/86
Z39.49 1985	Computerized Book Ordering	AP	0	S	Current

^a Status abbreviations: NS, new standard in process; AP, ANSI approved; RV, revision in process; RF, reaffirmation in process.

^b Activity: 0, none; 2, drafting standard; 4, ANSI standards action, 5, objections being considered.

^c Method: O, accredited organization; S, accredited standards committee.

ANSI maintains contact with some of the many regional groups that develop standards. CEN and CENELEC are among them. Eliminating technical barriers to trade are their major objectives. Both groups have arranged with ANSI to have their drafts listed in an institute periodical called *Standards Action* for public review. Through similar ANSI listings, U.S. exporters and other interests are given an opportunity to comment on draft ISO and IEC standards and on government regulations proposed by signatories to the GATT Standards Code. Members of ISO and IEC that are making major contributions to their work have an obligation to ensure that the international process is effective and timely and that not only is there useful cooperation between ISO and IEC but also with groups such as the CCITT of the ITU.

These standards-making bodies have developed standards for information processing, including cards and paper; character recognition; codes; data communication; data processing—vocabulary; data representation; disks, flexible; disks, rigid; flowcharts; languages; open systems interconnection; tapes, magnetic; tapes, paper; and office machines and supplies, to name a few.

ANSI, particularly Subcommittee X3, and the National Information Standards Organization Z39 (NISO) are working with ISO toward compatible international technical standards for a wide range of equipment, including computer and telecommunications equipment. ANSI Subcommittee X3 is responsible for most U.S. standards related to computing and data processing, whereas NISO is a private, not-for-profit organization in the United States that develops standards related to libraries, publishers, and information services. Standards developed by Subcommittee X3 and NISO are issued by ANSI. NISO is currently the secretariat for ISO, TC46/Subcommittee 4, which is responsible for standards in the area of library automation.

Table 4 identifies various formal standards-making bodies in the computer- and information-processing-related fields. The standards-making process will continue as technology evolves.

Table 5 shows a sampling of the various types of computer standards that are being, or have been, processed by ANSI.

BIBLIOGRAPHY

- American National Standards Institute, "ANSI's Role In International Standardization," American National Standards Institute, New York, May 1985.
- American National Standards Institute, "Guide to Submitting Standards to ANSI for Approval," American National Standards Institute, New York.
- American National Standards Institute, "1986 Progress Report," American National Standards Institute, New York, March 1986.
- American National Standards Institute, "Status Report of American National Standards Projects Assigned to the Information Systems Standards Board," American National Standards Institute, New York, March 17, 1986.
- Chuck Card et al., "The World of Standards," *Byte*, 8(2), 130-142 (February 1983).
- Hamilton, Robert W., "Nongovernmental Development of Regulatory Standards," *ASTM Standardization News*, 18-23 (January 1983).
- Hecht, Herbert, "Computer Standards," *IEEE Comp.*, 17(1) 33-42 (October 1984).

MacDonald, Joseph A., ed. dir., "Technology Update," *Inf. Disp.*, 1(9), 11
(September 1985).

Schweber, Bill, "Standards Guide Product Designs," *Comp. Des.*, 73-80
(July 1984).

M. E. L. JACOB

D. L. RINGS

APL FOR MICROCOMPUTERS

INTRODUCTION

APL is a very powerful, general-purpose interactive language that can be used in applications such as text processing, mathematical and scientific computations, and data base management. APL can be used both as a desk calculator, in which the programmer types the statements on a typewriterlike keyboard and then receives the response from the computer, and as a tool to write complex applications, where the user may not even be aware that APL is being used.

APL performs operations on ordered collections of data (arrays), as do assembly and high-level programming languages like BASIC, Pascal, and FORTRAN. But the novelty that APL introduces is that it treats arrays as single entities—in most cases without having to descend to the level of handling every element in an array separately—with which APL makes the programmer concentrate on the actual problem and forget about unnecessary details. For example, suppose we have two numerical matrices, *A* and *B*, with the same dimensions, *N1* and *N2*, and want to add the corresponding elements to form matrix *C*. If we were to program in assembly language, we would have to establish pointers to *A*, *B*, and *C*, set a register with *N1* and another with *N2*, and loop until all the elements were added. In BASIC, besides declaring the dimensions of *A*, *B*, and *C*, we would have to execute a set of statements such as

```
FOR I=1 TO N1:FOR J=1 TO N2:C(I,J)=A(I,J)+B(I,J):NEXT J,1
```

which is basically the same, although more neatly expressed. However, in APL, we would use the statement

```
C ← A + B
```

with no declarations. If we were to add two three-dimensional arrays, we would have to write a new assembler or BASIC program. The APL statement would be the same.

Another characteristic of APL is its interactivity. The user may sit at the terminal and write and debug the program at the same time. Sentences can thus be tested before being included in the defined functions (subroutines) that the user is writing.

APL is not difficult to learn, although the strange notation (at the beginning) may seem an unsurpassable barrier to many. It is not necessary to learn a lot of information before working in APL. Once logged on to the APL system, the student can start executing very simple statements and writing simple programs that can be made more and more complicated as the knowledge of the student increases.

In APL there is a clear separation between the language and the management of the environment. The language is a mathematical notation that can be used to describe an algorithm even without the help of a computer. The environment management represents the interaction between the algorithm, written in the APL notation, and the implementation of that APL notation in a computer, that is, the APL interpreter. The environment management is handled by the quad variables, quad functions, and system commands. In this way, the user can interact with the workspace (where all computations take place) both as a unit and with the different objects it contains.

APL can communicate with the outside world by means of shared variables. A shared variable is one that two systems can access and modify according to a pre-established set of synchronization rules. Shared variables are handled by the Shared Variable Processor (SVP). If an external system, whether hardware, (e.g., a printer) or software (e.g., a BASIC interpreter) is to be related to the APL system, an auxiliary processor has to be written. Communication between the APL system and the auxiliary processors is established via the SVP.

The advantages of the shared variable concept are readily understood: (a) the purity of the APL language is preserved, because it is not filled with an excessively large number of nonstandard primitives to handle every kind of peripheral and system that may be needed; (b) the management of every particular device is isolated from the rest, which facilitates the design and debugging capability of the APL system, in general, and every auxiliary processor, in particular; and (c) it is possible to access only the peripherals needed for an application and not to overload the computer memory with unnecessary processors.

APL is said to be so cryptic that even the author of a program cannot understand it after a certain period of time. This may be true if we consider that the power of APL allows the user to write both straightforward and well-documented programs, which are easy to comprehend by everyone, and long and intricate ones, which can be truly unintelligible due to their possible lack of documentation. But this is the fault of the user's programming style, not of APL. Obviously, the more powerful a language is, the more freedom it gives to the programmer to make cryptic programs but also to write complex and powerful applications. Which of the two tasks APL is dedicated to is not to be attributed to APL's power but to the programmer's will.

HISTORY

The name APL originated with the initials of a book written by K. E. Iverson, *A Programming Language*. Dr. Iverson first worked on the language at Harvard University and then continued its development at IBM with the collaboration of Adin Falkoff and others at the IBM T.J. Watson Research Center.

Since its publication in 1962, the APL notation was acknowledged as a revolution in the fields of mathematics and computing science. This opinion was confirmed in 1964, when a paper was published describing the IBM System/360 solely in terms of APL notation. The first implementation of the language was performed by IBM in 1965 and was first sold commercially in 1968. From that moment on, APL has been implemented in many computers, due to the growing interest people express in it.

To execute APL programs, it is necessary to write both an interpreter that runs in a particular machine and an APL supervisor consisting of a set of routines that interface the interpreter with the operating system of the machine.

Until 5 or 6 years ago, APL was only implemented in large machines, except for a few efforts to include it in minicomputers such as IBM 5100, IBM System 7, and IBM Series/1. The arrival of microcomputers and personal computers has resulted in a revolution in the computing world. Applications and programs that were supposed to work only in large machines were "transferred" to the small ones, giving the end user all the power of a large and time-shared computer in a small stand-alone machine. The next step for APL was obvious: APL systems were implemented in microcomputers.

To design an APL system for a microcomputer presents the following difficulties:

Implementing the APL System in a Computer with Little Main Storage

The APL system is fairly large. The interpreter alone occupies approximately 80Kb. The floating-point emulator and the SVP, together with the auxiliary processors that may be needed in a particular session, occupy around 20Kb. Finally, a sizable workspace of at least 64Kb is also needed. Therefore, at least 200Kb of main storage is needed to have a working APL system. Nowadays, most microcomputers provide much greater main memory, which is becoming less and less expensive. Five years ago this was not so. But this is not the only problem. The design of an APL workspace is greatly affected by the way the microprocessor handles its addressing space. The earlier ones, like the Z-80, were 8-bit addressing microprocessors. Others, like the Motorola 68000, have an address space of 32 bits. But the more common ones are those with 16-bit address space, like the Intel 8088, which is able to directly address a maximum of 64Kb of memory, called a segment, although the processor is provided with segment registers that allow it to access up to 1 million bytes. The inherent characteristics of an APL workspace require that the space it occupies in main storage be contiguous. Therefore, workspaces could not theoretically be greater than 64Kb. To solve the problem and dedicate all available memory in the machine to workspace, the designers of the IBM Personal Computer APL have applied a previous idea called the "elastic workspace." Roughly, the idea is as follows: The workspace is split into two parts: the main workspace, where all APL sentences are executed and objects are created and modified; and an extension called the elastic workspace, which can use all additional free memory. If space is needed for an operation in the main workspace, every object not currently being referenced will be automatically relocated to the elastic workspace and returned as needed. Instead of having all the interpreter loaded in main storage, other implementations have reduced the size of the code using, for example, overlays.

Nowadays, all IBM microcomputers (for instance, the IBM Personal System/2 models 30, 50, 60, and 80) come out with 1 Mb main storage and more, therefore, the workspace size is only limited by the operating system. The PC Disk Operating System can access a maximum of 640 Kb RAM memory (the rest, up to 1 Mb is dedicated to ROM, Graphics RAM, etc.). Thus, main memory can be assigned in the following way: (a) 40 Kb to PC-DOS, (b) 100 Kb to the APL interpreter, and (c) 500 Kb to the APL workspace, which make a fairly large workspace.

The Character Set

As we will see later, the APL character set is different from that of the other programming languages. The APL language has its own set of 135 characters, over one third of which are not included in the character set supported by most microcomputers. Because there are currently 256 characters, the maximum allowed, there is no room to add new ones. The first idea is to reduce the number of APL characters requiring substitution. In mainframes, APL supports the uppercase alphabetic character and the uppercase alphabetic character underline. The underline alphabetic characters have been substituted by the corresponding lowercase characters supported by all microcomputers. Some other characters have been replaced by existing ones that are very similar. Finally, some 30 characters had to be completely changed. This problem presents two questions: (a) which of the original characters of the microcomputer to substitute, and (b), how to output them in both displays and printers. Some implementations have solved both *a* and *b* by substituting keywords for the APL special symbols. Others have preserved the APL notation and have solved *a* by substituting the APL special characters for some of the less needed microcomputer characters. To solve *b*, several solutions have been presented.

Displays can be graphics and text. A graphics display usually has a software defined character set, and possibly several fonts can be loaded sequentially or in parallel. A font with the APL special characters included is then defined and used whenever the APL system is loaded. A text display presents the difficulty that the character set it supports is hardware defined, but, usually, the character set is defined in a removable chip. Implementations in microcomputers supporting this kind of display usually provide a chip with the APL character set defined as a substitute. Printers can also be graphics and text, and the problem of printing the APL character set has been solved in a similar way, with either a hardware or a software solution.

The Keyboard

When speaking of a keyboard we do not refer to the physical device but to the software that translates the keystrokes, either single or combined with shift keys, to the microcomputer internal character code (usually ASCII code). The differences between the keyboards lie in the fact that the character obtained when pressing a key or combination of keys does not have to be the same. APL has a keyboard of its own (see later in text). The earlier APL implementations used terminals with a very limited character set. Therefore, the number of different characters to be generated had to be reduced to a minimum. In APL, the 26 underline alphabetic characters plus an additional 19 characters were overstruck, that is, they were generated by printing two characters, one on top of the other. For example, Φ was generated by typing O, backspace, and I. Internally, the computer translated the three-character sequence, which generated the overstruck character. With the advent of the displays, overstruck characters could be directly typed and generated, so they become ordinary characters. However, in order to simplify the keyboard, that is, to dedicate a smaller number of keys to APL characters, some designers have revived the overstruck characters and have implemented them. This concept does not avoid the fact that the overstruck characters need to have an ASCII code assigned; anyway, some characters have to be substituted in the original character set both for displaying and for printing. For this

reason, other designers have not found it very useful and have implemented them as single keystroke characters. In any case, a keyboard program supporting the APL keyboard has been written and becomes active whenever the APL system is loaded. Some applications support both the U.S.A. and the National Keyboard programs while APL is active. A combination of keystrokes allows switching between keyboards. Finally, the physical microcomputer keyboards do not come with the APL characters engraved on the key caps. The problem has been solved by providing either stickers or a template with the layout of the APL keyboard.

Floating-Point Management

Microcomputers do not provide standard support for floating-point data. Usually they have either a mathematical coprocessor, which can be bought separately, or nothing. APL makes heavy use of double-precision, floating-point data. Thus, the solution has been either to fully support the floating-point coprocessor, when available, or to provide an emulator.

ELEMENTS OF THE APL LANGUAGE

The elements of the APL language are *data*, *primitive functions*, which operate on data, and *operators*, which act on primitive functions. Data, functions, and operators form *statements*, which have to comply with certain syntax and semantic rules to be executable. Otherwise, an error condition is raised. APL provides facilities for debugging and error recovery. Statements can be grouped to form *defined functions*. Functions (both its contents and the result of its execution) can be arguments to other functions.

An APL statement may be composed of (a) *label*, which must be followed by a colon; (b) an *expression*, which may be composed of other expressions; and (c) a *comment*, which must start with an **Ⓜ**. Each part is optional, but if more than one part is present, it must appear in the order given.

A comment is a nonexecutable piece of text. Everything in a statement to the right of the first comment symbol (**Ⓜ**) that is not part of a character constant is a comment. In some implementations, an expression must not be followed by a comment.

The left-most part of an APL expression must always be

- A name followed by the specification arrow (**←**). The name (variable) is assigned the value of the expression to its right.
- A branch arrow (**→**). This arrow serves to control the sequence in which the statements in a defined function are executed.
- Neither of the above cases. The result of the expression, if any, is then displayed.

When working interactively, the user is prompted to type the input indented six positions to the right, whereas APL answers are justified to the left.

A set of variables and defined functions can be assigned a name and stored in secondary storage to be used later. This is called a *workspace*. When in use, a workspace is said to be active and is located in the main storage. Part of each workspace is set aside to serve the internal workings of

the system, and the remainder is used to store APL objects and transient information generated during a computation.

Inactive workspaces are stored in *libraries*, where they are identified by names. They occupy space on disk and cannot be worked with directly. When required, an inactive workspace can be made active (loaded), or selected information can be transferred (copied) from an inactive workspace into an active one. Workspaces and libraries are managed by *system commands*.

The names of variables and defined functions used in calculations always refer to objects known by those names in the active work space; information about the progress of a program execution is maintained in the *state indicator* of the active workspace, and control information affecting the form of output is held within the active workspace.

In general, names of workspaces, functions, and variables, that is, identifiers, may be formed from any sequence of alphabetic and numeric characters that starts with an alphabetic character and contains no blank. They are subject to implementation dependencies.

The environment in which APL operations take place is limited to the active workspace. Hence, the same name may be used to designate different objects (i.e., functions or variables) in different workspaces without interference. Also, because workspaces themselves are never the subject of APL operations, but only of system commands, a workspace can have the same name as any object it holds.

Character Set

The APL characters belong to four main classes: alphabetic, numeric, special, and blank. The alphabet comprises the roman alphabet in uppercase and in lowercase, delta (Δ), and delta underline ($\underline{\Delta}$). The APL special characters are shown in Table 1.

APL also supports National and graphics characters, although they are not a part of APL's original character set.

The names in the table are for the symbols themselves and not necessarily for the functions that they represent. For example, the downstile (\lfloor) represents both the minimum, a function of two arguments, and the floor (or integer part), a function of one argument. In general, most of the special characters (such as +, -, \times , and \div) are used to denote primitive functions that are assigned fixed meanings, and the alphabetic characters are used to form names that may be assigned and reassigned significance as variables, defined functions, and other objects.

The space is a special character, used primarily as a separator. The spaces that one or more blank characters produce are needed to separate names of contiguous defined functions, constants, and variables. For instance, a blank is needed to separate the constants 3 (three) and 4 (four), otherwise the constant 34 (thirty-four) would be represented. Also, if F is a defined function and A a variable, we have to enter $F A$, otherwise the name FA would be understood by the computer. The exact number of spaces used in succession is not important, and extra spaces must be used freely. In general, spaces must be included whenever an ambiguity may arise. Spaces are not required between primitive functions and constants or variables, or between a succession of primitive functions, but they may be used if desired. For example, the expression $3+4$ may be entered with spaces, $3 + 4$.

The APL symbols can be classified according to the action they perform, as shown in Table 2.

TABLE 2 APL Symbols and Actions

Operation	Symbol	Monadic	Dyadic
<i>Functions</i>	Δ		Grade up
	∇		Grade down
	Φ	Reverse	Rotate
	Θ	Transpose	Transpose
	$+$	Identity	Plus
	$-$	Negation	Minus
	\times	Sign	Multiplication
	\div	Inverse	Division
	\lceil	Ceiling	Maximum
	\lfloor	Floor	Minimum
	$ $	Absolute value	Residue
	$!$	Factorial	Binomial
	$?$	Roll	Deal
	$*$	Power of e	Power
	\odot	Natural logarithm	Logarithm
	\circ	Pi times	Circular
	$=$		Equal to
	\neq		Not equal
	$<$		Less than
	\leq		Not greater
	$>$		Greater than
	\geq		Not less
	\wedge		And
	\vee		Or
	\wedge		Nand
	\vee		Nor
	\sim	Not	
	\boxtimes	Matrix inverse	Matrix divide
	ϵ		Member of
	ρ	Shape	Reshape
	\uparrow		Take
	\downarrow		Drop
	\lceil	Interval	Index of
	\leftarrow		Specification
	\rightarrow	Branch	
	$[;]$		Indexing
	\perp		Decode
	\top		Encode
	$,$	Ravel	Catenation
	$/$		Compression
	\backslash		Expansion
	\circ	Format	Format
	\circ	Execute	
<i>Operators</i>	\cdot	Outer product	Inner Product
	$/$	Reduction	
	\backslash	Scan	
	$[]$	Axis	

TABLE 2 (continued)

Operation	Symbol	Monadic	Dyadic
<i>Other</i>	<i>a</i>	<i>Comment</i>	
	<i>-</i>	<i>Negative sign</i>	
	<i>—</i>	<i>Underline</i>	
	<i>⎕</i>	<i>Unformatted input/output</i>	
	<i>⎕</i>	<i>Input/output</i>	
	<i>▽</i>	<i>Function definition</i>	
	<i>()</i>	<i>Delimiter parentheses</i>	
	<i>:</i>	<i>Label delimiter</i>	
	<i>space</i>	<i>Delimiter</i>	

The APL Keyboard

The APL language has a keyboard of its own, which has been preserved in the different implementations, except for the position of a few characters. The one represented in Figures 1, 2, and 3 corresponds to the IBM PC APL interpreter.

DATA

Data are produced by (a) explicit entry at the keyboard, (b) execution of APL functions or operators, and (c) use of shared variables and system variables.

Data may belong to three different classes: arrays, lists of arrays, and labels. An array is a set of numbers or characters (never mixed) disposed in an n -dimensioned latticed structure. An array consists of structure and contents. The structure is characterized by the number of axes or dimensions (*rank*) in it and the number of elements along each axis (*shape*). If the array is of rank zero, it is called a scalar; if it is of rank one, it is called a vector; if it is of rank two, it is called a *matrix*. The number of dimensions of an

Esc	1	2	3	4	5	6	7	8	9	0	+	×	Bkspc
Tab	Q	W	E	R	T	Y	U	I	O	P	←	⎕	Enter
Ctrl	A	S	D	F	G	H	J	K	L	[]	⋮	
Sh	/	Z	X	C	V	B	N	M	.	.	/	Shift	
Alt	Space											Lock	

FIGURE 1 Lower case APL keyboard.

Esc	~	-	<	≤	=	≥	>	≠	√	^	-	+	Bkspc
Δ	?	ω	ε	ρ	~	↑	↓	ι	ο	*	→	⌈	Enter
Ctrl	α	Γ	L	-	∇	Δ	ο	'	⌈	()	⌋	
Sh	⋄	C	⊃	∩	U	⊥	⌊		:	:	\	Shift	
Alt	Space												Lock

FIGURE 2 Upper case APL keyboard.

array is only limited by the implementation. An empty array has at least one dimension of value equal to zero. Such an array is either character or numeric but contains no elements. A scalar can never be empty.

The elements that form an array can belong to two different types: numerical or character. Numerical data can be internally stored in three different ways: Boolean, integer, or double-precision floating point. A Boolean array only contains ones and zeros. Each element is stored in a bit, and the structure is very compact. When every element in an array can be stored in a computer word, the array is said to be integer. Otherwise, it is called *floating*. The different numeric types can be freely mixed; APL itself takes care of conversions. *Characters* are stored in one byte each. Some implementations provide for 2-byte characters to support larger character sets, such as Katakana.

An array can be created in three different ways: as a constant, as a partial result, or as a variable.

A constant is a scalar or vector, either character or numeric, that appears explicitly in an APL statement. Constants are directly input at the terminal, and their values cannot change. A scalar numeric constant is a number entered by itself. A vector numeric constant is entered by listing the component numbers in order, separated by one or more spaces. A scalar character constant may be entered by placing the character between quotation marks; a vector character constant may be entered by listing no characters, two, or more characters between quotation marks. The system dis-

Esc	ι	⋄	⌈	⌊	⌈	⌊	⌈	⌊	⋄	⌈	⌊	⌈	Bkspc
Tab	q	w	e	r	t	y	u	i	o	p			Enter
Ctrl	a	s	d	f	g	h	j	k	l				
Sh		z	x	c	v	b	n	m					Shift
Alt	Space												Lock

FIGURE 3 Alternate APL keyboard.

plays such a vector as the sequence of characters, with no enclosing quotation marks and with no separation of the successive elements. The quotation mark character itself must be entered as a pair of quotation marks. Thus, the abbreviation of "cannot" is entered as 'CAN"T' and prints as CAN'T. All numbers entered or displayed are in decimal, either in conventional form (including a decimal point if appropriate) or in scaled form. The scaled form consists of an integer or decimal fraction called the multiplier, followed immediately by the character E, and an integer called the scale. The scale specifies the power of 10 by which the multiplier is to be multiplied. Thus, 1.44E2 is equivalent to 144. A negative number is represented by an overbar (¯) immediately preceding it. The overbar can be used only as part of a constant and is different from the sign that denotes negation (-).

A *partial result* is the result of the execution of an APL expression (e.g., 2×3).

The value of a *variable* may change. A variable is a name that can be assigned an array. The assignment statement has the form

name ← APL expression

The assignment is by value. This means that the name of the variable is assigned the result of the execution of the APL expression, not the expression itself. For instance, in the set of statements

```
A←10
B←A÷5
B
2
A←7
B
2
```

A is assigned the value 10, and B, the result of $10 \div 5$, or 2. Although A is later assigned a different value (7), the value of B is not modified.

A *list of arrays* is (in the computing sense of the word) composed of elements, which are arrays. The elements of a list of arrays are input separated by the symbol ';'. A list of arrays cannot be manipulated as an array is. It is used for indexing or indexed specification of arrays. The syntax of indexing is

name[array: .. ;array]

and the syntax of indexed specification is

name[array; .. ;array]←array

A label is a name, and its value is a numeric scalar. The label name is defined in a statement such as

Name: APL expression

The label name is assigned the value of the order this expression occupies in a defined APL function.

Functions

A function executes a special action on its arguments (if any) to produce a result (if any) that may be displayed on an output device or be used as an argument to another function.

Functions belong to three classes: primitive, derived, and defined. A primitive function is built in the APL system and cannot be changed by the user. A derived function is generated when an operator is applied to a primitive function. A defined function is created by the user and can be modified and erased; it is represented by a name.

PRIMITIVE FUNCTIONS AND OPERATORS

A primitive function can be monadic or dyadic. A monadic primitive function has one (right) argument. A dyadic primitive function has two arguments (right and left). They always generate a partial result. Primitive functions can be scalar or mixed.

Scalar Functions

Scalar functions are defined with scalar arguments, and their meaning is extended to nonscalar arguments, element by element.

Scalar monadic functions, applied to an array, produce another array of the same dimensions, the elements of which are the result of applying the function to every element in the argument. These functions are

+ - × ÷ | ⌈ ⌊ ? * ⅈ ○ ! ~

For instance, the *not* of the vector 1 0 1 (~1 0 1) is equivalent to applying the function ~ to every element of the vector.

```
~1 0 1
0 1 0
```

These functions only apply to numeric arguments. +A does nothing; -A changes the sign of every element of A; ×A returns the sign of the elements of A; ÷, |, ⌈ and ⌊ are, respectively, the inverse, absolute value, ceiling, and integer part. The symbol ? extracts pseudorandom numbers; * and ⅈ are, respectively, the power of e and the natural logarithm; ○ is the times of pi and, finally, ! is the factorial.

Scalar dyadic functions apply to two arguments with the same shape, or at least one of them with only one element. In this latter case, the only element is paired with every element of the other argument. These functions are

+ - × ÷ | ⌈ ⌊ * ⅈ ○ ! ∨ ∧ ∨* < ≤ = ≥ > ≠

For instance,

```
2 3 4 + 4 5 6
6 8 10
4 5 6 ÷ 2
2 2.5 3
```

The symbols $+$, $-$, \times , and \div are the arithmetic operations addition, subtraction, multiplication, and division. The symbols \uparrow , \lceil , and \lfloor represent remainder, maximum, and minimum. Power, logarithm, and circular are represented by $*$, \circ , and \circ . The latter symbol comprises 15 different transcendental functions among which are sine, cosine, tangent, hyperbolic sine, hyperbolic cosine, hyperbolic tangent, and the inverses. Finally, we have the Boolean functions, or (\vee), and (\wedge), nand (∇), and nor (∇), applied only to Boolean arguments and the relational functions, $<$, \leq , $=$, \geq , $>$, and \neq .

All dyadic scalar functions apply only to numeric arguments, with the exception of $=$ and \neq , which can be applied to character and to mixed arguments. Obviously, a character and a number are always different ($0=5='A'$).

Operators

When an operator is applied to a primitive function, a derived function is formed. The APL operators are axis, reduction, scan, outer product, and inner product.

Reduction, scan, outer product, and inner product can only be applied to scalar functions. The axis operator is a special case. Not only can it be applied to several mixed functions but also to functions derived from the operators reduction and scan.

Axis ($[]$) applies to the function that precedes it. A multidimensional array can be viewed as a collection of arrays of lesser rank that lie along any chosen axis. The axis operator is used to select the chosen axis and determines the direction of application of the derived function. The brackets enclose an expression yielding the index of the desired axis as a scalar or one-element vector. If a function that can be affected by the axis operator is applied to an array without the axis operator, the direction of application will be along the last axis.

Reduction ($/$) applies to the function that precedes it. If $V \leftarrow 2\ 3\ 4\ 5$, then $+ / V$ is the sum of the elements of V , and \times / V is their product:

$+ / V$	\times / V
14	120

In general, an expression of the form f / V is equivalent to the expression obtained by placing the symbol f between adjacent pairs of items of the vector V .

\uparrow / V	$2\ \uparrow\ 3\ \uparrow\ 4\ \uparrow\ 5$
2	2
$- / V$	$2-3-4-5$
-2	-2

Reduction of an empty vector by any function is the identity element of the function. For instance, $\times / 0\rho 0$ is one, whereas $\vee / 0\rho 0$ is zero.

If reduction is applied to an array without the axis operator, the direction of application is along the least significant dimension.

$A \leftarrow 3\ 4\rho\ 112$		
A	$+ / A$	$+ / [1]A$
1 2 3 4	10 26 42	15 18 21 24
5 6 7 8		
9 10 11 12		

The expression $f/[1]A$ is equivalent to $f\bar{A}$.

Scan (\backslash) applies to the function that precedes it. When the corresponding derived function is applied to a vector V , it produces a vector of the same shape, the I th element of which is equal to the corresponding reduction over the first I elements of V .

```

      +\2 3 4 5      v\0 0 1 0 1
2 5 9 14      0 0 1 1 1
      x\2 3 4 5      ^\1 1 0 1 0
2 6 24 120      1 1 0 0 0

```

If scan is applied to an array without the axis operator, the direction of application is along the least significant dimension.

$A \leftarrow 3 \text{ 4p } 12$

A	+\A	+\[1]A
1 2 3 4	1 3 6 10	1 2 3 4
5 6 7 8	5 11 18 26	6 8 10 12
9 10 11 12	9 19 30 42	15 18 21 24

The expression $f\backslash[1]A$ is equivalent to $f\bar{A}$.

Applied to certain Boolean and relational functions, scan is very useful to signal leading zeros or ones in a Boolean vector, or the first one or zero to the left. The symbol $v\backslash$ is used to convert to one all elements to the right of the first one; $\wedge\backslash$ is used to convert to zero all elements to the right of the first zero, $<\backslash$ is used to leave the first element one to the left and convert the rest of the elements to the right to zeros.

```

      v\0 0 1 0 1
0 0 1 1 1
      ^\1 1 0 1 0
1 1 0 0 0
      <\0 0 1 0 1 1 0
0 0 1 0 0 0 0

```

Outer product ($\circ.$) applies to any dyadic primitive scalar function, so that the function is evaluated for each member of the left argument paired with each member of the right argument. One of its uses is to build tables, like the ones used in elementary arithmetic texts.

$A \leftarrow 2 \text{ 3}$

$B \leftarrow 1 \text{ 2 3 4 5}$

$A \circ. \times B$	$A \circ. + B$	$A \circ. < B$
1 2 3 4 5	2 3 4 5 6	0 1 1 1 1
2 4 6 8 10	3 4 5 6 7	0 0 1 1 1
3 6 9 12 15	4 5 6 7 8	0 0 0 1 1

Inner product (\cdot) applies to the two functions that surround it. If P and Q are two vectors of equal length, the expression $Pf.gQ$ is equivalent to f/PgQ . The inner product extends to arrays other than vectors along the last axis of the left argument and the first axis of the right argument. The lengths of these axes must agree.

The inner product has multiple uses. For example, the expression $A+. \times B$ is equivalent to the arithmetic matrix product.

$M \leftarrow (14) 0. \leq 14$

M	$M+. \times M$
1 1 1 1	1 2 3 4
0 1 1 1	0 1 2 3
0 0 1 1	0 0 1 2
0 0 0 1	0 0 0 1

A common expression to remove leading blank columns in a character matrix is

$M \leftarrow 3 \uparrow \rho' \text{ ASDFG ZXCVB QWERT'}$

M
ASDFG
ZXCVB
QWERT

$(\vee \backslash' \vee. \neq M) / M$

ASDFG
ZXCVB
QWERT

Mixed Functions

Mixed functions may be applied to arrays of different shapes and produce results that may differ from its arguments both in shape and in type. They are grouped in five different classes: structural functions, selection functions; selector generators; numeric functions; and data transformations.

Structural Functions

The shape of an array can be determined by the monadic function called *shape* (ρ). This function, applied to an array A , produces a vector with the dimensions of A :

$A \leftarrow 2 \ 3 \rho \ 16$
A
1 2 3
4 5 6
 ρA
2 3

The expression $\rho \rho A$ calculates the rank of A . The rank of a scalar is zero, and its shape is the empty vector.

Reshape (ρ) restructures its right argument according to the shape expressed in its left argument. It is used to form arrays of more than one dimension. If the expression $Z \leftarrow L \ \rho \ R$ is executed, Z is an array of shape L , and the number of elements (taken from R) is the result of multiplying the values of the elements of L .


```

      2 3ρ1 2
    1 2 1
    2 1 2
      2ρ'ASD'
    AS

```

If any of the elements of L is zero, Z is an empty array. If L is an empty vector, Z is a scalar.

Ravel (ρ) produces a vector, the elements of which are those of the argument in row major order.

```

      A
    1 2 3
    4 5 6
      ,A
    1 2 3 4 5 6

```

Catenation (ρ) catenates vectors and scalars to form other vectors:

```

      V←1.2 3.4 5.6
      V,7 8 9
    1.2 3.4 5.6 7 8 9

```

Arrays are catenated along the least significant dimension.

If the function *catenate* is modified by means of the axis operator, arrays will be catenated along a dimension other than the least significant. The axis operator indicates which dimension.

```

      M1←3 2ρ'ABCDEF'      M1,[1]M2
      M2←3 2ρ'*O n u αω'    AB
      M1,M2                  CD
    AB*O                      EF
    CD n u                    *O
    EF αω                     n u
                               αω

```

Arguments with different types cannot be catenated.

Lamination is a kind of catenation that permits the creation of an array of rank higher than the ranks of the arguments. In this case, the axis operator must enclose a floating-point number indicating where the new dimension (2) should be inserted:

```

      X←'asdf',[0.5]'ASDF'      Y←'asdf',[1.5]'ASDF'
      X                          Y
    asdf                        aA
    ASDF                        sS
                                dD
                                fF
      ρX                          ρY
    2 4                          4 2

```

Monadic transpose (\Re) of an array inverts the order of the indices of its elements:

$A \leftarrow 2 \ 3 \rho 16$	$\Re A$
A	1 4
1 2 3	2 5
4 5 6	3 6

The arithmetic matrix transpose is expressed by $\Re M$.

Dyadic transpose (\Re) of an array permutes the indices of every element of the argument according to the vector left argument:

$A \leftarrow 3 \ 3 \rho 19$
A
1 2 3
4 5 6
7 8 9
1 1 $\Re A$
1 5 9
$A \leftarrow 2 \ 3 \ 5 \ 7 \rho 1210$
ρA
2 3 5 7
$\rho 2 \ 3 \ 1 \ 4 \Re A$
5 2 3 7

The expressions $\Re X$ and $(\Phi 1 \rho \rho X) \Re X$ are equivalent.

This function is very useful to interchange coordinates in a multidimensional array:

A	1 3 2 $\Re A$
abcd	aei
efgh	bfj
ijkl	cgk
	dh
mnop	
qrst	mqu
uvwx	nrw
	osw
	ptx

Reverse (Φ) inverts the order of the elements of a vector and the order of the columns of an array. Modified by the axis operator, it may reverse the order of the elements of any dimension in an array. The reverse along the most significant dimension, $\Phi[1]$, is also represented by the symbol Θ .

$\Phi 1 \ 2 \ 3$	
3 2 1	
$\Phi 2 \ 3 \rho 16$	$\Theta 2 \ 3 \rho 16$
3 2 1	4 5 6
6 5 4	1 2 3

$\Phi 2\ 3\ 4\rho\ 1\ 24$	$\Phi[2]2\ 3\ 4\rho\ 1\ 12$
4 3 2 1	9 10 11 12
8 7 6 5	5 6 7 8
12 11 10 9	1 2 3 4
16 15 14 13	21 22 23 24
20 19 18 17	17 18 19 20
24 23 22 21	13 14 15 16

Rotate (Φ) produces a result of the same shape as the right argument but with the elements cyclically rotated according to the left argument:

```

      2Φ1 2 3 4 5 6
3 4 5 6 1 2
      ~2Φ1 2 3 4 5 6
5 6 1 2 3 4
      M←3 4ρ 1 12
      M
1 2 3 4
5 6 7 8
9 10 11 12
      1 2 ~1ΦM
2 3 4 1
7 8 5 6
12 9 10 11

```

Rotate can also be modified by the axis operator to rotate an array along any dimension. The symbol $\Phi[1]$ is equivalent to Θ .

Rotate can be used to right justify a left-justified character matrix:

```

A←3 11ρ'Joe_____ChristopherRichard____'
A
Joe_____
Christopher
Richard____

(+/A≠'_')ΦA
_____Joe
_____Christopher
_____Richard

```

Selection Functions

All selection functions are dyadic. The right argument can be of any type. The left one (selector) is numeric, and it specifies the selection to be performed.

Take (\dagger) chooses from the right argument as many elements as the left argument indicates, from the beginning if the latter is positive and from the end if it is negative. If the number of elements to take is greater than the existing number, the result is padded by zeros if the argument is numeric and is padded by blanks if the argument is character.

```

      3+2 3 5 7
2 3 5
      -3+2 3 5 7
3 5 7

```

```

      6+2 3 5 7
2 3 5 7 0 0
      -6+2 3 5 7
0 0 2 3 5 7

```

```

      3+ABCD'
ABC
      2 3+3 4p 12
1 2 3
5 6 7

```

```

      (5+'ABCD'),'*'
ABCD *
      2 -5+3 4p 12
0 1 2 3 4
0 5 6 7 8

```

Drop (+) suppresses from the right argument as many elements as the left argument indicates, from the beginning if the latter is positive or from the end if it is negative.

```

      1 -1+3 1p 12
5 6 7
9 10 11

```

A very popular expression to designate the first dimension of an array A is $1 \uparrow A$, and $1 \downarrow A$ is used to designate the least significant dimension of a matrix.

Compression (/) is one of the most widely used APL functions. The expression U/X represents the compression of X by U . If U is a Boolean vector and X is another vector of the same dimension, U/X produces another vector comprised of the elements of X that correspond to ones in U .

```

      X+2 3 5 7 9
      U+1 0 1 1 0
      Y+U/X
      Y
2 5 7
      (~U)/X
3 9

```

There are multiple instances in which it is used. To suppress selected elements in a vector, for example, the spaces in a character string, we can do the following:

```

      A+ ' The APL Language'
      (A# ' ')/A
TheAPLLanguage

```

To suppress duplicate instances of elements in a vector,

```

      V+5 4 4 3 3 3 2 2 1 4 4 4 3'5 5 2 2
      ((1pV)=V 1V)/V
5 4 3 2 1

```

Expansion (\) is the opposite of compressing. If $Z \leftarrow U \backslash Y$, then U/Z is Y and $(\sim U)/Z$ is an array of zeros or spaces, depending on whether Z is numeric or character.

```
U\Y
2 0 5 7 0
```

```
Q←3 4ρ'ABCDEFGHijkl'
Q      0 1 1 0/Q      1 1 0 1 0 1\Q
ABCD      BC      AB C D
EFGH      FG      EF G H
IJKL      JK      IJ K L
```

If we want to substitute all instances of a character in a string by another, we can do the following:

```
A←' Substitution of Underline Characters in a string by blanks'
(A≠' ') \ (A≠' ') / A
Substitution of Underline Characters in a string by blanks
```

To test whether an array is numeric,

```
0=0\0ρX←1 2 3 4
1
0=0\0ρX←'CHARACTER'
0
```

Both compression and expansion can be modified by the axis operator. Compression along the first dimension is represented by the symbol \vdash and expansion along the first dimension is represented by \vee .

If X and I are vectors, the *indexing* of X by I is the selection of the I th elements of X .

```
X←2 3 5 7 9
X[1 3 5]
2 5 9
X[5 4 3 2 1]
9 7 5 3 2
```

The shape of $X[I]$ is the shape of I .

```
X←'ABCDEFG'
I←4 3ρ3 1 2 2 1 4 2 5 4 4 1 4
I
3 1 2
2 1 4
2 5 4
4 1 4
ρI
4 3
X[I]
CAB
BAD
BED
DAD
ρX[I]
4 3
```

If X is a matrix, it has to be indexed by a two-item list of the form $I;J$, where I selects rows and J , columns.

```

      M←2 4ρ3 1 4 2 1 4 4 1
      M
      M[2;3]
      M[1 2;3 1]
3 1 4 2      4      4 3
1 4 4 1      4 1

```

The shape of $M[I;J]$ ($\rho M[I;J]$) is $(\rho I), \rho J$. Both I and J can be arrays of any rank and shape. Its elements must belong to the set of possible indices of the dimensions of the array they are indexing.

$M[I;]$ selects all elements of the I th rows of M . $M[;J]$ selects all elements of the J th columns of M . $M[;]$ is identical to M .

```

      M[;M]
4 3 2 1
3 2 2 3

4 1 1 4
1 1 1 1

```

To index an n -dimension array the number of items in the list between brackets must be n , that is, the number of semicolons must be $n-1$. Therefore, a scalar cannot be indexed because it has no dimensions.

Indexed specification ($X[I]←Y$) is a special form of the assignment statement. The elements of Y are a substitute for the elements of X selected by the list I between brackets.

```

      X←2 3 5 7 9
      X[1 3]←6 8
      X
6 3 8 7 9

```

```

      M←2 4ρ3 1 4 2 1 4 4 1
      M
3 1 4 2
1 4 4 1
      M[1 2;3 1]←2 2ρ-1 4
~2 1 ~1 2
~4 4 ~3 1

```

Selector Generators

Index generator (\uparrow) applies to a one-element, non-negative integer array N and produces a vector of N number of elements, which contains the N first integers in ascending order.

```

      15
      1 2 3 4 5

```

If V is a vector and S is a scalar, $V\uparrow S$ produces the index of the first instance of S in V . If S is not in V , $V\uparrow S$ is $1+\rho V$. The shape of $V\uparrow A$, where A is an array, is ρA , and its elements, the indices of the elements of A in V .

```

      2 2 3 2 1 2 ~1
      1 5

```

```

V←'ABCDEFGHIJKLMNOPQRSTUVWXYZ '
ρV
27
V1'A*'
1 28
Z←V1'THE APL LANGUAGE'
20 8 5 27 1 16 12 27 12 1 14 7 21 1 7 5
V[Z]
THE APL LANGUAGE

```

To find the order of the left-most pair of differing elements in two vectors of equal lengths, *A* and *B*,

```

A←6 5 2 4 7 8
B←6 5 5 2 7 8
(A≠B)11
3

```

Membership (ϵ) produces a Boolean array of the same shape as the left argument. Any element of $L \epsilon R$ is one, if the corresponding element of *L* is anywhere in *R*, and zero otherwise.

```

1 2 3 4 5 6 7 ε 3 5
0 0 1 0 1 0 0

```

```

'ABCDEFGH' ε ' ALPHABET'
1 1 0 0 1 0 0 1

```

```

'S' ε 1 2 3
0

```

To test whether an array is empty,

```

0 ε ρ, A ← 2 0 ρ 1
1
0 ε ρ, A ← 2 3 ρ 1 6
0

```

Grade up (\uparrow), applied to a vector *V*, produces another vector with the same dimension as *V*, the first element of which is the index of the smallest element of *V*; the second is the index of the next smallest element, and so forth. Therefore, $V[\uparrow V]$ produces the elements of *V* in ascending order.

Grade down (\downarrow), applied to a vector *V*, produces another vector with the same dimension as *V*, the first element of which is the index of the greatest element of *V*; the second is the index of the next greatest element, and so forth. Therefore, $V[\downarrow V]$ produces the elements of *V* in descending order.

V←8 3 7 5 7	
$\uparrow V$	$\downarrow V$
2 4 3 5 1	1 3 5 4 2
$V[\uparrow V]$	$V[\downarrow V]$
3 5 7 7 8	8 7 7 5 3

Dyadic grade up (\uparrow), applied to a character array right argument R , produces the permutation of $\uparrow 1 \uparrow \rho R$, which puts the subarrays along the first axis of R in ascending order, according to the collating sequence in the left argument.

Dyadic grade down (\downarrow), applied to a character array right argument R , produces the permutation of $\downarrow 1 \downarrow \rho R$, which puts the subarrays along the first axis of R in descending order, according to the collating sequence in the left argument.

```
'ABCDE'⋄'DEAL'      'ABCDE'⋄'DEAL'
```

```
3 1 2 4              4 2 1 3
```

```
⊞←R←5 4ρ'dealDealdeadDeadDEED'
```

```
deal
```

```
Deal
```

```
dead
```

```
Dead
```

```
DEED
```

```
⊞←L←2 5ρ'abcdeABCDE'
```

```
abcde
```

```
ABCDE
```

<pre>⊞←Z←L⋄R</pre> <pre>3 4 1 2 5</pre> <pre>R[Z;]</pre> <pre>dead</pre> <pre>Deal</pre> <pre>deal</pre> <pre>Deal</pre> <pre>DEED</pre>	<pre>⊞←Z←L⋄R</pre> <pre>5 2 1 4 3</pre> <pre>R[Z;]</pre> <pre>DEED</pre> <pre>Deal</pre> <pre>deal</pre> <pre>Dead</pre> <pre>dead</pre>
--	--

Deal ($L \uparrow R$) produces a vector of length L , obtained by making L pseudorandom selections, without replacement from the population $\uparrow R$. $R \uparrow R$ produces a pseudorandom permutation of the items of R . For instance,

```
)CLEAR
```

```
CLEAR WS
```

```
A←'ABCDEFGH'
```

```
A[3?ρA]
```

```
BGE
```

```
A[(ρA)?ρA]
```

```
ECBGDhaf
```

Numeric Functions

Numeric functions apply to numeric arguments and produce numeric results.

Decode ($L \uparrow R$) yields the value of R evaluated in an L system number. If $L \leftarrow 24\ 60\ 60$ and R is a vector of elapsed time in hours, minutes, and seconds, $L \uparrow R$ is the elapsed time in seconds.

Encode ($L \uparrow R$) is, in general, inverse to decode.


```

10 10 10 10 1 1 7 7 6
1776
10 10 10 10 1 1776
1 7 7 6

```

To find the integer and fractional parts of a positive number,

```

0 1 23.45
23 0.45

```

Matrix inverse (\mathbb{H}), applied to a nonsingular matrix R produces the inverse of R . Matrix Divide ($Z \leftarrow L \mathbb{H} R$) produces a value of Z that satisfies the relation $L \div R = Z$, and is therefore the solution of the system of linear equations conventionally represented as $RZ = L$.

```

R ← (1 4) 0. 1 4
R       $\mathbb{H}R$        $R \div \mathbb{H}R$ 
1 0 0 0      1 0 0 0      1 0 0 0
1 1 0 0      -1 1 0 0      0 1 0 0
1 1 1 0      0 -1 1 0      0 0 1 0
1 1 1 1      0 0 -1 1      0 0 0 1

```

```

L ← 4 2 0 1 2 3 5 6 9 10 14
L
1 2
3 5
6 9
10 14

```

```

Z ← L  $\mathbb{H}R$        $R \div Z$        $(\mathbb{H}R) \div L$ 
Z
1 2      1 2      1 2
2 3      3 5      2 3
3 4      6 9      3 4
4 5      10 14     4 5

```

Data Transformations

Monadic format (\mathbb{F}) converts numeric data into its true alphanumeric representation.

For some kind of arguments, execute (\mathbb{E}) is the inverse to monadic format. It accepts a character vector and evaluates it as an APL statement. It is one of the most powerful APL functions because it allows the dynamic creation and execution of APL statements. System commands cannot be arguments to execute.

```

'A',  $\mathbb{F}$  25
A25
 $\mathbb{E}$  B ← 2 3 4
'B+2'
4 5 6

```

With \mathbb{E} , we can create variables dynamically.

```

X←8
Y←7
⊡'Δ',(⊖Y), '←X'
⊡'Δ',⊖Y
8
Δ7
8

```

We can also execute sentences dynamically, depending on a condition.

```

X←1
⊡(X/'A←1 2 3'),(¬X)/'A←4 5 6'
A
1 2 3

X←0
⊡(X/'A←1 2 3'),(¬X)/'A←4 5 6'
A
4 5 6

```

Dyadic format (\boxplus) accepts numeric left and right arguments and converts the right argument to a character array, according to the information given in the left argument.

```

R←3 2ρ12.34 ^34.567 0 12 ^0.26 ^123.45
ρ⊡←R
12.34          ^34.567
0              12
^0.26          ^123.45
3 2

ρ⊡←11 2 R      ρ⊡←12 ^2 R
12.34          ^34.57      1.2E001      ^3.5E001
.00            12.00      0.0E000      1.2E001
^0.26          ^123.45    ^2.6E^001    ^1.2E002
3 22              3 24

```

A special kind of dyadic format is the *picture format*, where the left argument, a character vector, is a pattern for the result.

```

'06/06/06'⊖1 2 86
01/02/86
' -551.45@⊖ ^1 0 10.1 ^100
-1 @ @ 10.1 @ -100 @
' 8555.50'⊖1 0 10 100
***1.00 ****.00 **10.00 *100.00

```

THE APL SYNTAX

An APL sentence consists of data, functions, and delimiters. Data can be constants and variables. Functions can be primitive, derived, and defined. A defined function can be dyadic, monadic, and niladic, with and without a

result. Monadic and dyadic defined functions, with a result, behave in the same way, syntactically, as primitive functions with the same number of arguments. A niladic defined function, with a result, is like a variable, syntactically. A defined function without a result behaves like the corresponding function with a result. A niladic function without a result must be the only object in an APL sentence.

In APL there is such a large number of primitive and derived functions that it would be very difficult to establish priority rules. To that we have to add the defined functions, the number of which is only limited by the user's needs or by the workspace size. Therefore, to simplify matters, it has been decided that in APL, both primitive, derived, and defined functions are treated alike, in the manner described below.

In conventional APL notation, the order of execution of an unparenthesized sequence of monadic functions may be stated as follows: The right argument of any function is the value of the entire expression to its right. For example, $\times L \div X$ means the sign (\times) of the integer part (L) of the inverse (\div) of X .

The same rule applies to the execution of dyadic unparenthesized functions: The right argument of a dyadic function is the value of the expression to its right. Its left argument is the value of the constant, variable, or niladic function (with result) immediately to the left. For instance, $10 \odot 10^{-3} O X$ is the decimal logarithm ($10 \odot$) of the sine (10) of the arc tangent ($^{-3} O$) of X . This rule can also be expressed in the following way: Every unparenthesized APL expression is evaluated from right to left.

```
2×3+4
14
4+3×2
10
```

Parentheses are used to control the order of execution in a statement. Any expression within matching parentheses is evaluated before applying any function outside the matching pair to the result. In the examples above, if we want to execute the function to the left previous to the function to the right, we must enclose the former and its arguments in parentheses.

```
(2×3)+4
10
```

A consequence of the rule is that the only concrete use of parentheses is to form the left argument of a function. For example, $(12 \div 3) \times 2$ is 8 and $12 \div 3 \times 2$ is 2. However, redundant pairs of parentheses can be used to help improve readability, and they do not provoke an error condition. Thus, the expressions $12 \div 3 \times 2$ and $12 \div (3 \times 2)$ are evaluated identically, with a result of 2.

There are, however, exceptions to this ideal situation.

Brackets, whether in indexing, axis operator, or indexed specification, are delimiter and primitive function at the same time. The syntax of indexing is the following:

```
array[list]
```

where list is a vector of arrays and is an expression, such as

array;array; .. ;array

To make the syntax of indexing consistent with that of the rest of the APL functions, the right bracket (]) should be omitted (its actual function is that of a delimiter). Only the left bracket ([) is necessary to represent the primitive function indexing. This also applies to the axis operator.

Indexed specification ($A[B] \leftarrow C$) is a function with three arguments (A, B, C), alien to the APL syntax. Such as it is now, the semantics of the expression " $A[B] f C$ " is context dependent: If f is the specification function, the elements in A marked by B are substituted by the correspondent elements in C , and the result of the expression is always C . On the other hand, if f is any other dyadic primitive or defined function, the left argument of f is the result of extracting the elements in A marked by B , and the result, in any case, corresponds to the execution of the expression.

Semicolon can be considered a very special kind of primitive function. It links two arrays or an array and a vector of arrays to form a list of arrays to index another array. However, because APL does not support lists, it treats them in a very rigid, although adequate, way to make the indexing syntax intelligible. Semicolon has been given minimum priority; it does not produce a partial result, and it can be dyadic ($A[2;3]$), niladic ($B[6;;]$), monadic with left argument ($C[7;1]$), and monadic with right argument ($D[;3]$). Actually, semicolon is only dyadic. A missing argument is implicitly the index generator of the corresponding dimension. For instance, $C[7;]$ is equivalent to $C[7;1(\rho C)[2]]$, and $B[6;;]$ is equivalent to $B[6;1(\rho B)[2];1(\rho B)[3]]$.

The axis operator always affects to some primitive functions, although it is not always explicitly shown, for example, $A \Phi [1] B$ means that rotate is performed along the first dimension of B ; here, the explicitly shown operator is $[1]$. However, in $A \Phi B$, rotate is performed along the last dimension of B ; here, the implicit operator is $[\rho B]$.

The syntax of operators is different from the syntax of functions. Functions are "data," operators are "functions," and derived functions are "partial results." The only dyadic operator is inner product (\cdot), and the monadic operators are outer product (\circ), reduction ($/$), scan (\backslash) and axis ($[]$). Inner product (\cdot) presents both functions on either side of the dot, like the syntax of a "dyadic function," for example, $+. \times$; however, the monadic operators have their only function argument to its left, for example, $\times/$.

The axis operators presents the syntax inconsistency intrinsic to brackets.

Outer product, despite being semantically monadic, is syntactically dyadic, with a dummy left function (\circ) with no use at all. To be consistent with the rest of the operators, the syntax of outer product would have to be $Ag.B$, or, even better, $Ag \circ B$, so as not to raise ambiguities with the decimal point, also represented by a dot.

In general, the APL syntax is ambiguous. Because there are no declarations, the class of an identifier is known only at execution time. This means that any name can be either a function, a variable, or a label at any time during an execution. For instance, the expression $A-B$ may mean (a) that the value of variable B is subtracted from the value of A or (b) that the value of variable B is negated and the result is passed as argument to the monadic defined function A . Even before executing the expression, A can be a variable and B a niladic function, and during the execution of B , A can be converted

into another function. Moreover, if the sentence is included in a loop, during one pass, *A* can be a function and *B*, a variable, and in the next pass, it can be reversed.

But the cases shown above are very far fetched, and in most situations, the classes of identifiers are kept throughout the execution.

DEFINED FUNCTIONS

Thus far we have explained a large set of very powerful primitive functions and operators that constitute the essence of the APL language. We can aggregate them to build powerful statements, but this is not enough. We need to aggregate those statements so that we can refer to them later on and avoid having to rewrite them. Otherwise, APL would be a powerful desktop calculator, nothing more. We need, in fact, to be able to define functions.

Arguments, Result, Local Names and Global Names

Defining functions is not enough, however. We may want to execute those functions with different values. For that, we may need to pass *arguments* to the functions and return a *result*. We may also want to define every function as a close unit, without producing side effects, its only communication with the rest of the functions being through its arguments. But the function may be fairly complex, and local names may have to be used, in addition to the arguments and the result.

Local names can be declared in a defined function. They are valid only during the execution of the function. When the function is ended, the names are automatically erased. Only the value of the result (if any) is returned. Those names that are not defined local to a function are *global* to it and to the functions called by the former whether directly or indirectly.

The scope of a local name is the function in which it is declared. The names declared local can be variables and functions. Labels are always local. A variable *V*, local to a function *F1* (let us call it *V(F1)*), is global to a function *F2*, called by *F1*, unless *F2* has declared a local variable with the same name *V*. In the latter case, the variable *V(F1)* is *shadowed* by the variable *V(F2)* upon entry to *F2*. *V(F2)* can change its value as many times as necessary. When *F2* is exited and execution returns to *F1*, *V(F2)* disappears and *V(F1)* recovers the value it had upon entry to *F2*. However, if *V* has not been declared local in *F2*, *V(F1)* will be global to *F2*, and the assignments to *V* performed in *F2* will affect it.

A defined function can be *monadic*, *dyadic*, and *niladic* (with no arguments) and may or may not have an explicit result. The number of arguments in a function is called its *valence*. Arguments are always passed by value. That means that arguments and result are always local variables. Upon entry into a function, the actual arguments are copied to the formal arguments. In this way, the formal arguments can be modified inside a function without changing the actual arguments. Upon exit, the formal result is copied to the actual result. Thus, it is possible in APL to build a function as a shielded structure that can be safely included in another function without introducing side effects to it. The scheme of a defined function is as follows:

```

      ∇header
[1] .....
[2] ..... body
[3] .....
[.] .....
      ∇

```

The six possible classes of defined functions differentiate from one another in the form of the *header*, according to the following table:

Number of Arguments	With Result	Without Result
Monadic (1)	$Z \leftarrow F R$	$F R$
Dyadic (2)	$Z \leftarrow L F R$	$L F R$
Niladic (0)	$Z \leftarrow F$	F

Every type of defined function may have local variables and functions. The local names are listed in the header and are separated by semicolons.

A dyadic function may be called either monadically (without the left argument) or dyadically. Thus, all defined dyadic functions are *ambivalent*, that is, the left argument is not required when the function is called. This feature can be used in cases where we want to assign the left argument a value by default. Suppose we want to simulate the transpose (\Re) APL primitive function, which is both monadic and dyadic. To do that, we can write the following defined function:

```

      ∇Z←A TRANSPOSE B
[1] →(0≠⊖NC 'A')/E
[2] A←⊖1ppB
[3] E:Z←A⊖B
      ∇

      TRANSPOSE 2 3 p16      ⍎ 2 3p16      2 1 ⍎ 2 3 p16
1 4      1 4      1 4
2 5      2 5      2 5
3 6      3 6      3 6

      1 1 TRANSPOSE 2 3 p16      1 1 ⍎ 2 3p16
1 5      1 5

```

In line [1] we ask if the left argument has been defined. If so, we jump to the line defined by the label *E*. Otherwise, the line [2], where *A* is assigned the default value is executed.

Branching

Thus far we can build a series of statements that can be executed one after the other. But we may want to execute those statements selectively according to the results of conditions, which are only known at execution time. Thus, we have to introduce some kind of control structure. The control structure in APL is represented by the primitive function *branch* (\rightarrow), which can be monadic and, exceptionally, niladic.

Sentences in a defined function are sequentially numbered, starting with one for the first statement in the function body, and continuing with successive integers as required. The numbering of sentences is automatic. Thus, $\rightarrow 3$ means the transfer of control to the sentence numbered three.

The branch function must always be the left-most function in a sentence. It does not produce a partial result. Its scope is the result of the APL expression to its right, which must be (a) an integer scalar; (b) a vector, the first element of which must be an integer; (c) an empty vector; or (d) nothing. In all other cases, an error results.

When the result of the expression is valid, the following rules apply:

- If the result is the number of a statement in the function, it represents the number of the sentence that will be executed next.
- If the result is an empty vector, execution continues with the next statement, if any, or execution ends.
- If the result is a number outside the range of statement numbers, execution ends. All negative integers and zero are outside the range of statement numbers for any function. A common way of exiting a function is $\rightarrow 0$.
- If the branch arrow is alone in a sentence, execution is exited of the function that contains the single branch arrow and all other functions that call to that function.

As we have seen, APL lacks the control structures that other languages have, such as IF=THEN=ELSE, DO=UNTIL, DO=WHILE, DO=END, and so forth. But APL does not need these structures. With branching and the APL primitives, such structures can be emulated easily. For instance, the structure "IF=X, THEN=ELSE" can be emulated in the following way:

```

      VZ←IF_THEN_ELSE X
[1]  →X/THEN
[2]  ELSE:Z←'The ELSE sentences have been executed'
[3]  →0
[4]  THEN:Z←'The THEN sentences have been executed'
      V

```

```

      IF_THEN_ELSE 1
The THEN sentences have been executed
      IF_THEN_ELSE 0
The ELSE sentences have been executed

```

If condition X holds, the result of sentence [1] is the value THEN, and execution jumps to the sentence with this label, which is executed. Else, the result of sentence [1] is the empty vector, and the execution continues with the statement labeled ELSE.

A very elegant way of emulating the IF=THEN=ELSE control structure is by means of the execution function (2):

```

      VZ←IF_THEN_ELSE X
[1]  THEN←'The THEN sentences have been executed'
[2]  ELSE←'The ELSE sentences have been executed'
[3]  2 (X/'THEN'), (~X)/'ELSE'
      V

```

```

      IF THEN ELSE 1
The THEN sentences have been executed
      IF THEN ELSE 0
The ELSE sentences have been executed

```

If $X=1$ (the condition holds), the result is the contents of THEN, which is executed; if $X=0$ (the condition does not hold), the result is the contents of ELSE, which is executed.

This emulation is elegant but expensive in execution time. It is not recommended unless the contents of the sentences to be executed are only known at execution time.

Recursion

A recursive function is a defined function, the name of which has not been declared local and that calls to itself.

A classical example of a recursive expression is factorial. The factorial of an integer number X , represented in arithmetic notation by $X!$ is $X! = X \times (X-1)!$, for $X > 1$, and 1 for $X=1$ and for $X=0$. In APL it can be expressed as follows:

```

      ∇ Z ← FACTORIAL X
[1] → (X > 1) / E
[2] Z ← 1
[3] → 0
[4] E: Z ← X × FACTORIAL X-1
      ∇

```

Recursion is a very useful tool but a very expensive one, because it uses much of the APL system resources. In the FACTORIAL function above, the function has to be called as many times as the value of X indicates. Every time the function is called, a new set of names X , Z , and E , appear, which shadow the older ones, and when the function is exited, the new names disappear, the result is returned, and the shadowed names recover their values. The user does not have to worry about the housekeeping, because the system maintains an execution stack to keep track of the functions that are in execution, of pairs of parentheses open and not yet closed, of pairs of brackets open and not yet closed, and so forth. The size of the stack is limited and implementation dependent. It is for this reason that a poorly designed recursive function can fill the stack very easily and provoke an error condition called STACK FULL. Besides, a function not only represents an overhead in stack space but also in time. Iteration can sometimes substitute for recursion.

The FACTORIAL function written using iteration is as follows:

```

      ∇ Z ← FACTORIAL X
[1] Z ← 1
[2] E: → (X ≤ 1) / 0
[3] Z ← Z × X
[4] X ← X-1
[5] → E
      ∇

```


(A shorter way of calculating the factorial of a number X other than a negative integer in APL is $!X$.)

Function Definition

There are several ways to set up a defined function in a workspace: (a) A previously defined function can be brought into a workspace using the commands `)LOAD` or `)IN`; and (b) a function can be created using the system functions `⎕FX` and `⎕CR`, or using the built-in `del (V)` editor that every APL system includes. (Some implementations also include a full screen function editor.)

All defined functions have the same structure, regardless of the way they have been established in a work space.

FUNCTION EXECUTION, INPUT/OUTPUT, ERRORS, AND DEBUGGING

The first step in the design of an application is the design of the functions that will form it. An APL application is never wholly coded before the testing stage starts. Due to APL interactivity, the usual way of writing an application is by doing it piecemeal—line by line, until a function is completed and function by function until the application is finished. This procedure increases the programmer's productivity greatly. When the application is put together, despite the fact that every function has been tested separately, errors appear that have to be located and corrected. This is the debugging stage. Finally, the application has to recover from two types of error conditions: (a) undetected failures in its own design, and (b) inconsistencies in the user input data.

Function Execution

A monadic or dyadic defined function (it should be remembered that a niladic function is syntactically like a variable) is like a primitive function in all aspects, except that it cannot be argument to an operator to form a derived function. A defined function can call other defined functions and can even call itself (recursion).

When a function is called, the following actions take place: (a) Formal arguments are created and assigned the value of the actual arguments. In all other aspects they are treated like local variables. (b) Global variables with the same name as the local variables are shadowed. (c) Local variables are created. Execution starts with the first statement and continues with the successive statements, unless execution is diverted by branch statements.

Input/Output

Sometimes, during the execution of an application, it is necessary to ask the user for some data, and the application must be able to read these data and continue executing. APL provides mechanisms for input and output that are represented by the symbols `⎕` (quad) and `⎕` (quote-quad).

Evaluated input is represented by the symbol `⎕`, appearing anywhere other than immediately to the left of the specification symbol. In this case, the symbol is interpreted as data. When this symbol is encountered in an

execution, the symbols \square : are displayed, the terminal opens, and the system waits for input. This input is evaluated, and its result is placed as a substitute for \square in the executing line. For example,

```

      ∇ Z←F X
[1]   Z←X,  $\square$ 
      ∇
        A←2
        F 3
 $\square$ :
      A
3 2
      F 4.5
 $\square$ :
      6×10
4.5 60

```

Character input is represented by the symbol \square , appearing anywhere other than immediately to the left of the specification symbol. In this case, the symbol is interpreted as data. When this symbol is encountered in an execution, the terminal opens, and the system waits for input. This input is entered as is, accepted as characters, and placed instead of \square in the executing line. For example,

```

      X← $\square$ 
O'CLOCK
      X
O'CLOCK

```

Normal output is represented by the symbol \square placed to the left of the specification symbol. It then indicates that the result of the expression to the right of the arrow is displayed in the normal APL format. A quad symbol is syntactically equivalent to a variable, and it may appear in the same places as a variable. For instance,

```

      X← $\square$ ←7+ $\square$ ←4-5
-1
6

```

It is very useful to include the symbols $\square←$ at appropriate places in a statement, during the debugging stage, to check the values of partial results.

Normal output includes an end-of-line character, so that the next data entered or displayed at the terminal will start at a new line. However, bare output ($\square←$) does not include the end-of-line character if it is followed by character input (\square).

```

      ∇ Z←ENTRY;A
[1]   A← $\square$ ←'Enter Input: '
[2]   Z←(ρA)← $\square$ 
      ∇

```

```

X←ENTRY
Enter Input: INPUT ENTERED
X
INPUT ENTERED

```

Halted Execution

The execution of a function may be interrupted for several reasons: for an error condition, for an attention signal, or for a stop control. In this case, the function is said to be *suspended*. Execution can be resumed by entering a branch statement from the terminal. Execution is quitted by entering a single branch symbol (→) from the terminal. When a function is suspended, the function (if any) that has called it, is also interrupted; the function that called the latter is interrupted too, and so on. These interrupted functions are said to be in *pendent* state.

When a function is suspended, the name of the function is displayed together with the number of the line in which the execution was stopped. If the stop is due to an error condition or to an interrupt, the line is also displayed with an appropriate message and a mark signaling the point of interrupt. Unless a specification has been performed in this line before the interrupt has taken place, the state of the computation is restored to the situation just before the line in question has begun to execute.

When a function is suspended and the terminal opens, all kind of APL sentences can be executed as usual, but one fact must be taken into account: The variables local to the suspended function are active, and the shadowed variables are inactive. Therefore, the environment may be different to the one with no suspended functions.

The system command,)SI, displays the state indicator. For instance,

```

)SI
* F[3]
  G[2]
    H[5]

```

This example means that execution was interrupted while executing statement 3 in function *F*; function *G* calls function *F* in line 2, and function *H* calls function *G* in line 5. The symbol * to the left of function *F* means that this is the interrupted function, and *G* and *H* are pendent. Execution of *G* and *H* cannot be directly resumed, unless execution of *F* is resumed.

If a new execution is started and later suspended while function *F* is suspended, the state indicator may look like the following:

```

. )SI
* I[4]
  J[7]
* F[3]
  G[2]
    H[5]

```

The most recently suspended function always appears first.

The line counter is a system variable, □LC, that holds the current statement number of the lines that are executing. In the example above, the value of □LC would be

□LC
4 7 3 2 5

A suspended function can be edited with the del (▽) editor, for instance, to correct the error that provoked the suspension. If the suspended function is edited in such a way that the number or the values of labels, or the number of sentences, are modified, the state indicator is *damaged*, and execution cannot be resumed. It must be exited, by typing a single branch arrow (→). Execution can be resumed later. Pending functions can also be edited. In the example above, after typing →, the state indicator would look like the following:

```

      →
    )SI
* F[3]
  G[2]
  H[5]

```

A new → would completely clear the state indicator.

The state indicator can be cleared in one single operation using the command)RESET.

If the interrupt is not due to an error or the editing does not damage the state indicator, execution can be resumed by typing →□LC, equivalent to (in the above example) →3 2 5, which means go to line 3 in function F and resume execution.

Error Condition

In APL an error condition may arise for multiple reasons. All errors provide consistent messages. Errors can be classified into (a) those referring to the language itself, and (b) those relating to the environment.

The errors referring to the language are DOMAIN, INDEX, LENGTH, RANK, SYNTAX, and VALUE.

- DOMAIN error arises when an argument is not valid, for example, 2+'A'.
- INDEX error arises when the indices between brackets are out of range, for example, (15)[9].
- LENGTH error occurs when the shapes of the arguments are not conformable, for example, (15)×17.
- RANK error occurs when the ranks of the arguments are not conformable, for example, (2 3 ρ16)÷4 5 6.
- SYNTAX error appears when the syntax of the sentence is not correct, for example, 2+ε7.
- VALUE error appears when reference is made to a name that has not been defined.

Errors relating to the environment differ with the implementation. The most common are DEFINITION, □...IMPLICIT, INTERRUPT, SI DAMAGE, STACK FULL, SYMBOL TABLE FULL, SYSTEM, SYSTEM LIMIT, and WORK-SPACE FULL.

- DEFINITION occurs when the function definition editor is not correctly used.
- `⋄...IMPLICIT` occurs when the referred quad variable is not properly set.
- INTERRUPT is not really an error but a warning. It means that the interrupt key has been pressed.
- SI DAMAGE arises when the state indicator has been damaged.
- STACK FULL means too many nested functions.
- SYMBOL TABLE FULL means that too many symbols are used.
- SYSTEM means that there has been an irrecoverable error in the system, the workspace is cleared, and the session is lost.
- SYSTEM LIMIT, an implementation limit appears, for example, the maximum number of elements in an object has been reached.
- WORKSPACE FULL arises when there is no space available in main storage to continue working.

The system commands also give warnings; for example, if a workspace is searched in the wrong library, the system responds with the message WORKSPACE NOT FOUND.

Error Recovery and Debugging

A programming error may or may not provoke an error condition. In the first case, the function is suspended in the line where the error is apparent. In the second case, the function executes correctly, but the result is wrong. Both cases can be considered the same, and the debugging techniques that will be explained later apply equally to them.

When an error condition arises in an APL line, the execution stops and the wrong line is displayed, together with a mark pointing where the error has appeared. For instance,

```

F
SYNTAX ERROR
F[4]
  Z←1-X
  ^

```

The execution of function *F* is now suspended, and the terminal is open, waiting for the user to type in a new sentence. The user may then be in two situations: (a) If the error is limited to the line, the line can be changed and execution resumed by typing `→⋄LC`. For instance, in the example above, the user may notice that there is a typing error. It should say `Z←1-X` instead of `Z+1-X`. The offending symbol is substituted, and execution is resumed. (b) The error has been made apparent in that line, but it has been produced elsewhere, but whereabouts? To discover where the error has actually been generated could be a long and tedious task if we had only the listing of the code to rely on. Fortunately, APL systems come with a set of debugging aids that make the task much easier. The debugging aids are quad output, tracing, stopping, and alternate execution. Suppose we want to debug a function that drops all blank rows in a character matrix. The function we write is

```

      ∇ Z←DR BL X;U
[1]   U←∇≠ X≠' '
[2]   Z←U≠ X
      ∇

```

However, when this function is executed, we get an error.

```

      A←3 4ρ'good   bad '
      A
good
bad

      DR BL A
LENGTH ERROR
DR BL[2]
      Z←U≠ X
      ^

```

In this particular case, LENGTH error means that the length of vector *U* and the first dimension of *X* are different.

Quad output ($\square\leftarrow$) can be included at various points in the APL statements that form a function to display the partial results obtained at those points. For instance,

```

      ∇ Z←DR BL X;U
[1]   U←∇≠ □≠X≠' '
[2]   Z←U≠ X
      ∇

      DR BL A
1 1 1 1
0 0 0 0
1 1 1 0
LENGTH ERROR
DR BL[2]
      Z←U≠ X
      ^

```

The result displayed above (elements of *X* different from blanks, equal to 1) gives us no extra information. More information is needed.

Another debugging tool is the *tracing* feature. Any desired line in any defined function can be traced; that is, the result of a traced line can be displayed and identified by the function name and line number. A trace is established in certain lines of a function by assigning the numbers of those lines to the name of the function preceded by the symbols *TΔ*. For example,

```

      TΔDR_BL←1
      DR_BL A
1 1 1 1
0 0 0 0
1 1 1 0
DR_BL[1]
1 1 1 1
LENGTH ERROR
DR_BL[2]
      Z←U÷X

```

Here, we have obtained the partial result $\square \div X \neq ' '$ and the value of U , which is the result of line 1. Now we begin to notice something absurd. The value of U is supposed to be a Boolean three-element vector, with each element indicating whether a row in matrix X is all blanks (1) or not (0). However, U is a four-element vector in which all elements are 1. So the error lies in line 1. The right part of the sentence, $X \neq ' '$, is correct; we have already seen the result. So, the fault must lie in the left part. Thus isolated, the part where the error must be is very easy to find. Because we want to discover which rows are all blanks, we must reduce the matrix $X \neq ' '$ along the second dimension, not along the first. Therefore the definite design of the function is

```

      ∇ Z←DR_BL X;U
[1]   U←∇/X≠' '
[2]   Z←U÷X
      ∇

```

By now, the state indicator must be quite crowded. To clear it, the command)RESET must be executed.

Traces can be changed from within and from without a defined function. To turn the traces off, we execute

```

      TΔDR_BL←0

```

If this function is executed, the result is

```

      DR_BL A
good
bad

```

Stopping is another way of error debugging. Execution stops before the previously designated lines. For example,

```

      SΔDR_BL←1 2
      DR_BL A
DR_BL[1]

```

Execution has stopped just before line 1 of function DR BL. The user is now free to interrogate (or change) the values of any variables, such as

```

      X
good
      bad
      U
VALUE ERROR
      U
      ^

```

Stopping can be combined with tracing. Both trace and stop may be reset at any time.

Stop is the most powerful debugging tool of the three explained so far. Because the function can be stopped at any line, variables and partial results can be examined at will, and functions can be edited on the spot. Again, if the state indicator is damaged after the modification of a function, execution can be quitted by typing a single arrow (\rightarrow); afterwards, execution can be restarted. If the state indicator is not damaged, execution can be resumed by entering a right arrow followed by the line number to be executed. To resume at the next line, the common expression $\rightarrow \square LC$ is used, meaning "branch to the first line number kept in the line counter $\square LC$." For example,

```

       $\rightarrow \square LC$ 
DR_BL[2]

```

The system function *execute alternate* ($\square EA$) is an extremely useful way of error recovery. It is particularly useful in preventing possible errors in the input provided by the user when executing an application. For example, if the user inputs a number instead of a character, an error condition will arise and execution will stop.

```

      'A',  $\square$ 
 $\square$ :
      2
DOMAIN ERROR
      'A', 2
      ^

```

Both arguments of $\square EA$ are character strings holding APL statements. The right one is executed first. If no error occurs, the left argument is not executed; but if the right statement fails, the left is executed. Rewriting the above statement,

```

      '"A", ⚡ B'  $\square EA$  '"A", B  $\leftarrow \square$ '
 $\square$ :
      2
A2

```

The left statement may contain a branch expression to transfer control in the case of an error. A useful technique to prevent errors from a user written function is to invoke it in the right statement and include the error recovery routine in the left argument; $\square EA$ will then start the latter if any error occurs during the execution of the former.

SYSTEM FUNCTIONS AND SYSTEM VARIABLES

The APL system variables have been included to make it possible for APL to interact with the environment of the system in which APL operates. This can be done within the language by identifying certain variables as elements of the interface between APL and the operating system and using these variables for communication between them. In the same way, system functions are used to serve as interface between the APL system and the environment.

System functions and variables are denoted by keywords starting with the symbol \square . They cannot be erased. System variables can be localized. They are described below.

Canonical representation (\square CR), applied to a character vector representing the name of a function, converts the latter into a character matrix in which each line is a row of the matrix. For instance,

```

      ▽ Z←FACTORIAL X
[1]  →(X>1)/E
[2]  Z←1
[3]  →0
[4]  E:Z←X×FACTORIAL X-1
      ▽

```

```

      ρ□←□CR 'FACTORIAL'
Z←FACTORIAL X
→(X>1)/E
Z←1
→0
E:Z←X×FACTORIAL X-1
5 19

```

Function establishment (\square FX), applied to a compatible character matrix, converts it to a defined function. If the operation is successful, the result of \square FX is the name of the newly established function. For instance,

```

      □FX □CR 'FACTORIAL'
FACTORIAL

```

Both \square CR and \square FX can be used to create defined functions.

Delay, (\square DL) causes a pause in the execution of the statement where it appears. It can be used to establish timings where accuracy is not of primary importance.

Execute alternate (\square EA) is a key system function for error trapping and recovery in APL applications. If the statement $L \square$ EA R is executed and there is an error in R or R is interrupted, the execution of R is finished without any warning and L is executed instead. The execution of L is subject to normal error handling. For instance,

```

      '2+3' ⍳EA '4×5'
20
      '2+3' ⍳EA '4÷0'
5
      '2÷0' ⍳EA '4÷0'
* DOMAIN ERROR
      2÷0
      ^

```

Expunge (⍳EX) is used to erase APL objects dynamically.

Name Classification (⍳NC) accepts a name and returns a number indicating the class of that name, that is, whether that name corresponds to a label, a variable, a function, or is not assigned and is free to be used.

Name List (⍳NL) is the opposite of ⍳NC. Applied to a class number, ⍳NL returns a character matrix, in which each row represents the name of an object in the workspace belonging to that class. This function is very useful when we want to deal with functions and variables dynamically. For instance, if we want to erase all functions and variables in a work space, we execute

```
⍳EX ⍳NL 2 3
```

But if we want to erase only those functions and/or variables that meet certain requirements, for instance, those that start with an 'X' we execute

```
⍳EX ('X'=A[;1])÷A÷⍳NL 2 3
```

⍳NL can be used in conjunction with ⍳CR to generate the definitions of all or certain functions automatically, for instance, to print them.

Peek/poke (⍳PK) is the most powerful of the system functions because it has access to main storage, can read and write anything, anywhere, and can also execute object code programs written previously by ⍳PK.

Transfer form (⍳TF) has a dual function. (a) Applied to a variable or a function name, it converts the object to a character vector, which is the transfer form of that object. (b) Applied to a compatible transfer form, the corresponding object is established in the work space. For instance,

NUMERIC+2 3p16	CHARACTER+3 3p'QWERTYASD'
A←⍳TF 'NUMERIC'	B←⍳TF 'CHARACTER'
A	B
N NUMERIC 2 2 3 1 2 3 4 5 6	C CHARACTER 2 3 3 QWERTYASD
)ERASE NUMERIC)ERASE CHARACTER
NUMERIC	CHARACTER
VALUE ERROR	VALUE ERROR
NUMERIC	CHARACTER
^	^
⍳TF A	⍳TF B
NUMERIC	CHARACTER
NUMERIC	CHARACTER
1 2 3	QWE
4 5 6	RTY
	ASD

```

      V Z←Y TIMES X
[1]   Z←Y×X
      V
      □CR'TIMES'
Z←Y TIMES X
Z←Y×X
      □TF'TIMES'
FTIMES 2 2 11 Z←Y TIMES XZ←Y×X

```

All system variables can be specified, but some of them pass the information assigned to them to the APL processor, whereas others completely ignore it. The variables that pass the information assigned to them are described below.

Comparison tolerance (□CT), or fuzz, is internally used in those primitive functions that require comparison of two values, like monadic ⌈ and ⌊ and dyadic <, ≤, =, ≥, >, ≠, |, ε, and 1. It is included because floating-point elements cannot be stored with more than a certain number of significant digits (implementation dependent). In this way, expressions such as 0.9999999999999999 and 3×3 can be considered equal.

Format control (□FC), is a five-element character vector used as implicit argument for picture format. It allows such things as the substitution of the decimal comma for the decimal point.

```

      □FC[1]←','
      '5.5555'⌘3.1416
3,1416

```

To define a replacement character other than blank,

```

      □FC[3]←'#'
      '855555'⌘1234
##1234

```

To fill a field that is too small for a number or a nonscalar element in picture format,

```

      □FC[4]←'?'
      '55555'⌘123456 1234
????? 1234

```

Index origin (□IO) determines the order of the first element in a vector. By default it is always one, but some applications require it to be zero. Affected are monadic and dyadic 1 and ?, ⌈, ⌊, indexing, the axis operator, and the left argument of ⌘. For instance,

$\square \text{IO} \leftarrow 1$	$\square \text{IO} \leftarrow 0$
$\square \leftarrow A \leftarrow 0.1 + i5$	$\square \leftarrow A \leftarrow 0.1 + i5$
1.1 2.1 3.1 4.1 5.1	0.1 1.1 2.1 3.1 4.1
A[1 3 5]	A[0 2 4]
1.1 3.1 5.1	0.1 2.1 4.1
$\square \leftarrow B \leftarrow 2 \ 3p'qweasd'$	$\square \leftarrow B \leftarrow 2 \ 3p'qweasd'$
qwe	qwe
asd	asd
C \leftarrow 'zxc'	C \leftarrow 'zxc'
$\square \leftarrow D \leftarrow B, [1]C$	$\square \leftarrow D \leftarrow B, [0]C$
qwe	qwe
asd	asd
zxc	zxc
1 10D	0 00D
qsc	qsc

The *latent expression*, an ordinary APL expression assigned to $\square \text{LX}$, is automatically executed whenever the workspace is loaded. It is used to start an application without user intervention.

Printing precision ($\square \text{PP}$) determines the number of significant digits to be printed to the right of the decimal point.

Printing width ($\square \text{PW}$) determines the width of the printed listings.

Random link ($\square \text{RL}$) is the base for the generation of random numbers in dyadic and monadic \cdot . Every time a random number is generated, $\square \text{RL}$ is changed by the system.

The system variables that ignore the information assigned to them are described below.

Account information ($\square \text{AI}$) returns a vector containing the user identification, computer time, connect time, and keying time.

Atomic vector ($\square \text{AV}$) gives all APL character values.

Line counter ($\square \text{LC}$) contains a vector of line numbers of defined functions, either currently being executed or halted, with the most recently activated line number first. The way to restart a suspended function is $\rightarrow \square \text{LC}$.

Time stamp ($\square \text{TS}$) returns year, month, day, hour, minute, second, and millisecond.

Terminal control ($\square \text{TC}$) gives a three-element character vector containing back space, new line, and line feed.

Work area ($\square \text{WA}$) gives the amount of storage that is still available in the workspace. It can be employed in a program, for instance, to choose between two alternative algorithms: a fast algorithm, but one that occupies more storage than another, slower one.

SYSTEM COMMANDS

System commands control the management of workspaces as units and of selected parts of them. They can only be invoked by individual entries from the keyboard and cannot be executed dynamically as part of a defined function or the execute primitive function. They are represented by a keyword preceded by a right parenthesis. Some commands may execute two actions, depending on their syntax. For example, $\cdot \text{WSID}$ alone returns the current name of the

active workspace ;)WSID followed by a valid work space name changes the name of the active workspace .

System commands belong to three groups: Those dealing with workspace storage and retrieval, those relating to the active workspace , and the one that provides access to the operating system.

Following are commands dealing with workspace storage and retrieval.

-)WSID is used both to ask for the name of a workspace and to give it the name with which it can be saved later.
-)SAVE stores a workspace in secondary storage, to be loaded later.
-)LOAD activates a workspace that was stored before with the command)SAVE.
-)OUT is used to store a workspace , or selected items of it, in transfer form in a file. Later, these items can be brought back to the active workspace using the)IN command.
-)DROP deletes a workspace from secondary storage.
-)LIB obtains a list of the files stored in the indicated library.

The command relating to the active workspace are as follows:

-)CLEAR resets the active workspace to its initial state.
-)SYMBOLS either sets the size of the symbol table or gives an indication of the degree of its occupancy.
-)STACK either sets the size of the execution stack or returns its size.
-)ERASE is used to delete specific objects from the active workspace .
-)IN brings back objects that were stored previously in transfer form to the active workspace, using the)OUT command.
-)RESET clears the state indicator (the execution stack).
-)FNS displays a list with the names of the defined function in the active workspace .
-)VARS displays a list with the names of the variables in the active workspace .
-)SI displays the state indicator, showing the status of the halted functions, with the function most recently halted first.
-)SINL displays the state indicator in the same way as)SI, but with every function listed, showing the list of names that are local to it.
-)OFF, the command to return to the system, ends APL and gives control back to the operating system.

Commands control workspace management and cannot be executed dynamically. It seems as though workspaces can only be handled while sitting at the terminal, but that is not so. The problem is alleviated by two facts: (a) Some commands can be emulated and (b) an auxiliary processor, AP101, has been built to execute all kind of APL lines from an input stack when the terminal opens. Both facts make it possible for the designers to write applications in such a way that the user may not even be aware that APL is used. Some commands that can be emulated are listed below.

-)OUT and)IN use the [TTF system function and the file management auxiliary processor.
-)DROP uses the file management auxiliary processor.

-)CLEAR and ERASE use the □EX and the □NL system functions.
-)FNS and)VARS use the □NL system function.
-)WSID uses the □PK system function.

SHARED VARIABLES

Thus far, language, system variables, system functions, and system commands have been described. Implementors try to make the language implementation independent. With that aim, the APL ISO standard is being created by an international group of people belonging to different corporations. Not only is this a philosophical necessity for maintaining a clean language but also for the very practical reason of making the applications transportable between different implementations. However, system variables, system functions, commands and, most of all, the shared variable processor are heavily implementation dependent.

Shared variables constitute a means of interchange of information between two concurrent processors. In particular, variables may be shared between an APL workspace and another processor that is a part of the APL system to obtain a variety of effects such as the control of peripheral devices or the handling of other systems.

IBM first introduced the concept of shared variables and has adhered to it in the various implementations of the APL language that have been made for the different IBM machines. The shared variable concept and the auxiliary processors have several advantages, namely, (a) the language is kept as is; (b) only the auxiliary processors needed at a given moment occupy space in main memory—there is no need to have everything loaded; (c) in the special case that it is necessary to handle an experiment for which there is no auxiliary processor, the APL system provides the users with information and macros to tailor auxiliary processors to suit their own needs.

Not all implementors of APL interpreters have included the shared variable processor. Some have substituted a fairly complete set of quad variables to handle all kinds of peripherals for it. Some of the cases not covered by the quad variables could be handled by writing machine code programs that could be executed by the □PK system function (or the equivalent in the corresponding implementation).

An APL shared variable can be global or local and is syntactically indistinguishable from an ordinary variable. A shared variable can be referenced, that is, accessed, by either of the processors that share it. An accessed variable can be assigned (set) and referred to (used). A processor can share variables with several other processors, but a variable can only be shared by two processors.

The actions necessary to handle the sharing of variables are four: offer, access, retraction, and inquiry and are managed by the system functions □SVO, □SVC, □SVR, and □SVQ, respectively.

To share a variable with another processor, the sentence to execute is

```
p □SVO n
```

where *p* is the identification of the second processor, and *n* is a character vector containing the name of the variable to be shared. The result of the expression above is the degree of coupling of the name *n*, zero if no offer

has been made, one if an offer has been made but not matched, and two if the variable has been accepted and the sharing is completed.

The monadic function \square SVO is used to ask for the degree of coupling of a variable.

The sharing of a variable between two processors can be left constraint free, that is, either processor can set and use the variable at any time, or it can be controlled by the processors. In most practical applications it is necessary to know that a variable has been assigned a value between two successive uses or that use has been made of an assigned value. Therefore, an access control mechanism has been provided.

The access control works by inhibiting the setting or use of a shared variable by either of the owners, depending on the accessing protocol, which is set jointly by the two owners, using the dyadic form of the system function \square SVC. The right argument of \square SVC is a character vector containing the name of the shared variable, and the left argument is a four-element Boolean vector, as specified below.

- If the first element is one, the second processor must access the shared variable before the first one can assign it.
- If the second element is one, the second processor cannot assign the variable until the first one has accessed it.
- If the third element is one, the first processor cannot read the variable twice unless the second one has assigned it.
- If the fourth element is one, the second processor cannot read the variable twice unless the first one has assigned it.

Sharing is a symmetric operation; neither processor has precedence over the other. The resulting access protocol is determined by the logical *or* (\vee) of both access requests. The dyadic function \square SVC returns an explicit result that is the new setting of the access protocol combining the specifications of both processors; however, monadic \square SVC returns the access protocol as seen by the processor that issues it.

The offer of a variable can be retracted by the monadic function \square SVR applied to the name of the variable. A variable is automatically retracted if it is erased or if a new work space is loaded. If a shared variable is local to a defined function, it is automatically retracted when the execution of the function is ended (this is obvious because the variable is erased).

The monadic function \square SVQ is used by a processor to determine the numbers of processors that offer variables to it and the names of those variables.

AUXILIARY PROCESSORS AND APPLICATIONS

In its two versions, 1.0 and 2.0 (both of them designed at the IBM Madrid Scientific Center, Spain), the IBM PC APL includes a fairly large set of auxiliary processors (mainly in version 2.0). The IBM PC APL Interpreter has been designed in an operating systemlike fashion to make it possible to access and execute all kinds of systems and APL and nonAPL programs from APL and communicate with other computers. Applications using the auxiliary processors have also been included and are described below.

One auxiliary processor, AP2, has been designed to get from the APL workspace special areas of memory, called partitions, where programs can be dynamically loaded and executed and nonexecutable programs can be

loaded. The type of programs that can be executed are (a) auxiliary processors (therefore, it is not necessary to load the ones used in a given session in advance); (b) exchange—assembly programs, that is, programs capable of accepting data from APL and returning a result; and (c) standard DOS (IBM PC disk operating system) programs to which data may be passed through files. Programs and files can later be unloaded, and partitions freed and returned to the APL workspace.

Several workspaces have been built using the facilities that this auxiliary processor provides: (a) a full screen defined function editor which loads an IBM PC full screen editor, like the Personal Editor; (b) a utility work space which, by executing exchange assembly programs, is able to perform otherwise long and painful tasks in a very short time, such as finding all appearances of a string in a very long character vector (APL does not provide a primitive for it); (c) a workspace to call and execute Professional FORTRAN subroutines.

AP80 has been written to control the IBM graphics printers and to print selected parts of a session dynamically without user intervention.

The Stack and Profile Auxiliary Processor, AP101, allows the execution of APL lines, commands included, when the terminal opens, providing a way to load APL automatically, execute an application, and disconnect APL. It also allows function key definition, library definition, and screen and keyboard mode change.

AP103 allows the handling of the BIOS (basic input/output system) and DOS interrupts and the management of the input/output ports. A workspace has been built to emulate DOS commands like CHDIR, CLS, DATA, DIR, ERASE, MKDIR, RENAME, RMDIR, TIME, VERIFY, VOL, and some functions that DOS is lacking. Another workspace uses BIOS interrupts to control the keyboard, the APL interrupt keys, or a joystick or to read a key.

The Full Screen Display Management Auxiliary Processors AP124 and AP205 allow screen-handling functions such as write, read, print, color management, and trapping of keyboard interrupts. They are essential to add professional appearance and increase productivity to APL applications. Building menus and editors is one of their large set of applications.

The Graphics Auxiliary Processor, AP206, allows the design of graphics in a graphics display. It also allows the definition of graphics in main memory (without being displayed) to be later printed or displayed in sequence at varying time intervals to produce animation. It represents graphic information defined as a set of vectors, drawing the boundaries and/or filling the area. With it, workspaces to create text foils and/or pie charts, or build histograms and graphic plots of mathematical functions have been written. Also, a graphics editor has been built to create, modify, and delete complex images composed of text, plots, and graphics.

The 3278/79 Emulation Card Communications Auxiliary Processor, AP190DCA, is designed to permit the APL interpreter to communicate with a host session. It can be used (a) to run IMS/CICS sessions to collect and process data in the IBM PC and (b) to run laboratory equipment and pass results to host session.

The Asynchronous Communications Auxiliary Processors AP232 and AP232X can be accessed from APL on the IBM PC and provide an interface for communications between the IBM PC and a host (IBM System/370 and compatibles). An application has also been built to support communications with IBM Virtual Machine Facility/370 (VM/370) on an IBM System/370 (or equivalent

machine) with an ASCII port: This work space permits the use of the IBM PC as an intelligent APL and non-APL workstation.

The GPIB/IEEE-488 Auxiliary Processor AP488 is an interface to permit an IBM PC to act as an IEEE-488 controller/talker/listener via the IBM GPIB support software and associated hardware. A work space has also been built to provide cover functions to access most of the functions provided by the IBM GPIB support software.

The DOS File Auxiliary Processor AP210 is used to read from or write to fixed-length files under control of the DOS file system. The reading and writing can be either sequential or random. Besides the DOS file management, it can be used to emulate (a) those APL system commands dealing with files, such as)IN,)OUT, and)DROP, and (b) some DOS commands like COMP (compare), TYPE, and RENAME. Another use of this auxiliary processor is the emulation of complex array management. (A complex array is the one whose elements are arrays themselves instead of being scalars.)

The Music Generator Auxiliary Processor, AP440, provides an easy way to create music through the attached loudspeaker.

Not included in the IBM PC APL Interpreter version 2.0, but compatible with it, is another auxiliary processor, AP999, which includes a subset of a PROLOG-like interpreter and, therefore, makes a PROLOG-like inference processor available to APL. An application workspace has also been written that contains a set of APL functions to facilitate the communication between APL and the auxiliary processor. The workspace can define and delete facts and rules, ask questions, and load and save facts and rules from and into DOS disk files.

Other applications written in IBM PC APL are

- The control of a computerized greenhouse, which required a special auxiliary processor for digital and analogic signal input/output.
- Text processing in which APL characters are intermixed with ordinary text.
- Compilers and interpreters in which execution speed is not of primary importance.
- For continuous simulation; to act as a developing system for a newly designed computer; to design and test algorithms that are later implemented in compiled or assembly languages; to run a couple of local area network models, one of Ethernet type, and the other a token ring; to implement an algorithm for the solution of the two-body problem in celestial mechanics; to calculate trajectories of satellites; in algebra, geometry, statistics, graphics, electronic circuit network calculation, computer-assisted instruction, medicine, and so forth.

APL COMPILATION

As we have seen before, APL is a very powerful language. It allows the design of a great deal of applications, although some of them are very slow due to the interpretative character of the language. Because every APL line is to be syntax analyzed before being executed, those applications requiring heavy use of loops have to spend plenty of time analyzing the same lines every time they are executed. (Loops do not always mean slow execution time. Some primitive functions, like \uparrow and ∇ , are expensive in execution time and are sometimes substituted advantageously by certain iterative algorithms.)

The execution of programs written in a high-level language (source code) requires a translator to make them intelligible to the computer. Translators belong to two main classes: interpreters and compilers. An interpreter accepts a program written in a high-level language and executes its statements as soon as it finds them. It does not generate any code. A compiler accepts a program written in a high-level language and converts it into an equivalent program written in a lower-level language (object code).

A translator consists of a set of passes that can be executed sequentially or in parallel; the order can never be inverted.

1. The lexical analyzer converts the string of characters that form the source code sentences into a string of tokens. Tokens are names, constants, primitive functions, delimiters, and so forth.
2. The syntax analyzer checks that the sequence of tokens in the source code conforms to the language rules.
3. The optimizer recognizes idioms that can be interpreted very easily. This pass is optional.
4. Finally, in an interpreter, the last pass is the execution of the code, and in a compiler, the generation of the object code.

The reason the syntax analysis of interpreted APL programs has to be postponed to execution time is because the APL syntax is inherently ambiguous. In most applications, however, the semantics of the APL code is kept invariant during the execution, making it unnecessary to delay parsing until execution time.

To date, there have been several proposals for speeding up the APL interpretation. Some investigators propose the resolution of syntax ambiguities by parsing a whole workspace (data flow). Others propose a partial compilation of the parts of the code that can be sped up, leaving the rest to the interpreter. Others intend to treat APL functions like PL/I or FORTRAN subroutines, with separate compilation, declarations both of the classes of names, and of the valence of functions and restrictions in the branch sentences, so as to know at compile time which sentences are a target for branching (this allows more extended code optimization and register allocation). In general, a common agreement is the suppression of `⌈FX`, `⌈CR`, `⌈EX` because they modify the environment at execution time. For the same reason `⌈` should be restricted to converting strings of characters to numerals. It would be advantageous to leave the compiled APL language compatible with the interpreted one. In this way, an application could be designed and debugged using the interpreter facilities before being compiled. Also, if the designers of APL compilers adhere to the standard and if the APL compilers accept input in a common form, for example, transfer form, APL compilers could be independent of any APL interpreter implementation (within the frame of a particular machine) and everyone could benefit from them.

APL2

Since its appearance in 1962, all kind of extensions have been requested from APL, possibly due to the fact that it allows such ease of programming that the user soon becomes accustomed to it and begins asking for new features with which to do more with even less effort. Some of these extensions, for example, the primitive functions `execute` and `format`, have already been

implemented in all APL systems. But the stronger requirements for extensions have referred to the field of data structures. People want to be able to work with sets of arrays in the same way as they are now able to work with sets of scalars (arrays). Once the need to dispose of higher-level structures has been signaled, several solutions have been proposed in which the new objects are called arrays of arrays, incomplete arrays, generalized arrays, nested arrays, lists, trees, and so forth. A theory of arrays has been developed by Trenchard More.

The addition of new structures to APL has to be performed in such a way that the original language is not changed so that applications running in APL can be migrated to the new language.

APL2 has been designed as an answer to those requirements for extensions. APL2 supports arrays of arrays. An array can now be recursively defined as an n -dimensional latticed structure of which the elements themselves are arrays. In an array, numbers and characters can now be mixed.

A new numeric data type has been included in APL2, the type *complex*. Complex arithmetic is achieved by considering all numbers to be elements of the complex number field and defining all arithmetic on complex numbers. A complex number constant may be represented in three ways: real and imaginary parts, separated by the letter *J*; magnitude and angle in radians, separated by the letter *R*; and magnitude and angle in degrees, separated by the letter *D*.

Several new primitive functions and operators have been added and others already existent have been extended to support the arrays of arrays and the new type.

Operators can now be defined and applied to defined functions with which the utility of primitive and defined functions has been greatly enhanced.

The system management has been extended with the addition of several new system variables, functions, and commands. The defined function editor has also been enhanced.

APL2 is an international programming language where commands can be entered and system messages can be received in a large set of different national languages.

VSAPL workspaces are easily migrated to APL2.

APL2 has been successfully applied in the fields of graphics (CDDM), among others, relational data bases (SQL/DB2), and representation and manipulation of mathematical formulae.

APL2, as such, has not been implemented in any microcomputer yet. However some APL implementations, like Dyalog APL and APL*PLUS, support nested arrays.

CONCLUSION

APL is an interactive and powerful computer language that has been successfully implemented in several microcomputers by many companies. APL is used in a great variety of fields. The set of applications of APL seems to be limited only by the programmers' imagination. But APL is not the ultimate language. APL2 comes next. In the near future, many APL2 implementations will appear on microcomputers. What is beyond APL2? Already, proposals for extensions to APL2 have been made.

The trend of the APL languages is, on the one hand, toward a greater flexibility and power in the language, and on the other, toward making it

possible for APL to interface with a growing number of software and hardware systems, giving those systems the possibility of communicating with one another through APL. A microcomputer will then become (it is beginning to be now) a wonderful workbench for all types of research.

BIBLIOGRAPHY

Books

- APL—An Introduction, Independent Study Program, SR20-7183, IBM Corporation,
 APL Language, GC26-3847, IBM Corporation,
 APL Programming Guide, G320-6735, IBM Corporation,
 APL2 Programming. Language Reference, SH20-9227, IBM Corporation,
 Brown, J. A., "A Generalization of APL," Ph.D. Dissertation, System and Information Science, Syracuse University, 1971.
 Fifth Working Standard for Programming Language APL, International Standard Organization, Document Number ISO TC97/SC5 WG 6 N38, June 30, 1983.
 FinnAPL Idiom Library Finnish APL Association FinnAPL, Helsinki, 1982.
 Franksen, O. L., Mr. Babbage's Secret. The Tale of a Cypher—and APL, Strandbergs Forlag, Birkerød, Denmark, 1984.
 Gilman, L., and A. Rose, APL: An Interactive Approach, 3rd ed. Wiley, New York, 1983.
 IBM Personal Computer APL Interpreter, 6024077, Version 1.0, IBM Corporation,
 IBM Personal Computer APL Interpreter, 6391329, Version 2.0, IBM Corporation,
 IBM Personal Computer Disk Operating System, 6024211, IBM Corporation,
 IBM Personal Computer Technical Reference, 6936895, IBM Corporation,
 Iverson, K. E., A Programming Language, Wiley, New York, 1962.
 Iverson, K. E., Algebra: An Algorithmic Treatment, Addison-Wesley, Reading, MA, 1982.
 McGrew, J., Introduction to APL2, SH20-9229, 1983.
 Personal Editor, 6024051, Version 1.0, IBM Corporation,
 Polivka, R., and S. Pakin, APL: The Language and Its Usage, Prentice-Hall, Englewood Cliffs, NJ, 1975.
 Professional FORTRAN, 6138815 and 6362003, IBM Corporation,
 Turner, J., APL Is Easy! An Introduction to APL, STSC, Inc., 1982.

Papers

- Abrams, P. S., "An APL Machine," Stanford University, Stanford, CA, 1970.
 Abrams, P. S., "What's Wrong with APL?" ACM APL/75 Congress, Pisa, Italy, 1977.
 Alfonseca, M. and M. L. Tavera, "APL on Small Machines," SEAS Spring Meeting, Noordwijkerhout, 1983.
 Alfonseca, M., "APL, Graphics and Artificial Intelligence," APL et l'Ingenierie Informatique, AFCET, Paris, December 1985.

- Berry, M., "Shared Functions and Variables as an Aid to Applications Design," APL84 Conference Proceedings, ACM, Helsinki, Finland, 1984.
- Brown, J. A., "Evaluating Extensions to APL," ACM STAPL/Sigplan Proceedings APL79 Conference, 1979. Rochester, NY
- Brown, J. A., "APL2 Syntax--Is It really Right to Left?" APL83 Conference Proceedings, ACM, Washington, D.C., 1983.
- Budd, T. A., "An APL Compiler for the UNIX timesharing system," APL83 Conference Proceedings, ACM, Washington, D.C., 1983.
- Budd, T. A., "An APL Compiler for a Vector Processor," ACM Transactions on Languages and Systems, Vol. 6, 1984.
- Canet, J., "An Example of APL's Incursion in the Medical World. The Usefulness of this Language for the Noncomputer Physicians," APL84 Conference Proceedings, ACM, Helsinki, Finland, 1984.
- Dickey, A. K., "Improving APL Performance with Custom Written Auxiliary Processors," APL85 Conference Proceedings, *APL Quote Quad*, 15, 4 (1985).
- Falkoff, A. D., K. E. Iverson, and E. H. Sussenguth, "A Formal Description of System/360," *IBM Syst. J.* 3, 3 (1964).
- Falkoff, A. D., and K. E. Iverson, "The Design of APL," *IBM J. of Res. Devel.* 17, 4 (1973).
- Falkoff, A. D., "An APL Standard," ACM-STAPL/Sigplan Proceedings APL79 Conference, 1979. Rochester, NY
- Falkoff, A. D., "One Man's View of the Ideal APL System," APL84 Conference Proceedings, ACM, Helsinki, Finland, 1984.
- Flury, J. and D. Lankhorst, "APL Application Development in a Micro System Environment," APL84 Conference Proceedings, ACM, Helsinki, Finland, 1984.
- Foster, G., "On the Locus of Program Flow within and among Secondary Functions," ACM-STAPL/Sigplan Proceedings APL79 Conference, 1979. Rochester, NY
- Gates, H. P., "A Microcomputer APL*PLUS System Implementation," APL83 Conference Proceedings, ACM, Washington, D.C., 1983.
- Ghandour, Z., and J. Mezei, "General Arrays, Operators, and Functions," *IBM J. Res. Devel.* 17, 4, (July, 1973).
- Gull, W. E., and M. A. Jenkins, "Recursive Data Structures in APL," *CACM*, 22, 4 (February, 1979).
- Iverson, K. F., "APL Syntax and Semantics," APL83 Conference Proceedings, ACM, Washington, D.C., 1983.
- Johnson, R., "The Dynamic Incremental Compiler of APL\3000," ACM-STAPL/Sigplan Proceedings APL 79 Conference, 1979. Rochester, NY
- Lathwell, R. H., "Some Implications of APL Order of Execution Rules," ACM-STAPL/Sigplan Proceedings APL79 Conference, 1979. Rochester, NY
- Lathwell, R. H., "A Formal Description of APL," *IBM Philadelphia Scientific Center Report* 320-3008, November 1971.
- Miller, T., "Tentative Compilation: A Design for an APL Compiler," ACM-STAPL/Sigplan Proceedings APL79 Conference, 1979. Rochester
- More, T., "Axioms and Theorems for a Theory of Arrays," *IBM J. Res. Devel.* 17, 3 (1973).
- Morrow, L. A., "APL Standardization," APL83 Conference Proceedings, ACM, Washington, D.C., 1983.
- Myrna, J. W., "Increasing the Use of APL," APL83 Conference Proceedings, ACM, Washington, D.C. 1983.

- Polivka, R. P., "The Impact of APL2 on Teaching APL," APL84 Conference Proceedings, ACM, Helsinki, Finland, 1984.
- Saal, H. J., and Z. Weiss, "A Software High Performance APL Interpreter," ACM-STAPL/Sigplan Proceedings APL79 Conference, 1979. Rochester, NY
- Searle, J. R., "The Future of Programming—Whither APL?" APL 84 Conference Proceedings, ACM, Helsinki, Finland, 1984.
- Tavera, M. I., and M. Alfonseca, "Elastic Work Space for an APL System," IBM Technical Disclosure Bulletin TDB04-81, No. SZ8800004, 5147-5148, April 1981.
- Tavera, M. L., M. Alfonseca, and J. Rojas, "The IBM Personal Computer APL System," APL84 Conference Proceedings, ACM, Helsinki, Finland, 1984.
- Tavera, M. L., M. Alfonseca, and J. Rojas, "An APL System for the IBM Personal Computer," *IBM Syst. J.* 24, 1 (1985).
- Tavera, M. L., "An APL Test Compiler, Written in APL," 1985. (unpublished)
- Weigang, J., "An Introduction to STSC's APL Compiler," APL'85 Conference Proceedings, *APL Quote Quad*, 15, 4 (1985).
- Weiss, Z., and H. J. Saal, "Compile Time Syntax Analysis of APL Programs," *APL Quote Quad*, 12 (September 1981).
- Wiedman, C., "Steps Toward an APL Compiler," ACM-STAPL/Sigplan Proceedings APL79 conference, 1979. Rochester, NY
- Weidman, C., "A Performance Comprison between an APL Interpreter and Compiler," APL83 Conference Proceedings, ACM, Washington, D.C., 1983.

Periodicals

- APL Market News, Springer, P.O. Box 503, IJmuiden, The Netherlands.
- APL Quote Quad, ACM, 1133 Avenue of the Americas, New York, New York 10018.
- Vector, British APL Association, 13 Mansfield Street, London.

MARIA TAVERA

APOLLO COMPUTER, INC.

Apollo Computer, Inc., pioneered the design and development of high-performance workstations for technical professionals and was the first in the computer industry to manufacture and market these powerful computer systems.

In 1980, the year the company was founded, Apollo introduced the concept of networked technical workstations, a milestone in high-performance computing and the successor technology to traditional approaches to information processing. Apollo created a computing environment that, for the first time, delivered the computational power of dedicated minicomputers and the advanced data access and communications capabilities of time-sharing systems.

Trends in computer technology over the preceding 20 years led to the development of the company's DOMAIN System network in 1980. Advances in technology, including the development of microprocessors and random access memory based on integrated circuits, resulted in the evolution of minicomputers and superminicomputers that offered computing power comparable to earlier mainframe systems at a fraction of the price. Concurrently, the use of computers evolved from batch processing by a few users on large central mainframes to time-sharing, in which a number of relatively low-cost terminal devices are connected to the central computer system's computing resources. Because of their reduced cost, minicomputers could be dedicated to a single user or task and permit a high degree of user interaction. However, these dedicated minicomputers did not easily allow users to share programs, data, or peripheral devices. Conversely, time-sharing systems did permit groups of users to share these resources, but the responsiveness and interactive features of time-sharing systems are impaired as the number of users increases.

Apollo's DOMAIN System local area network, for the first time, combined the power and high degree of interactiveness offered by dedicated minicomputers with the economy of shared system resources of time-sharing systems by interconnecting independent 32-bit superminicomputers. The company developed three key elements to provide this capability: virtual memory distributed over the network rather than localized in only one computer; high-speed, high-resolution graphics features extending the range of applications for DOMAIN System networks; and specialized software that directs and controls all network activities. These elements enabled each workstation to handle large and demanding applications, such as those in the engineering and scientific areas, at a small fraction of the price of a mainframe computer.

Throughout the early and mid-1980s, Apollo maintained a constant record of innovation and technological leadership. In 1981, Apollo introduced the first commercially available technical workstation, the first network-wide virtual memory and transparent, distributed file system, and the first multivendor display environment. In 1982, Apollo introduced the first color workstation, and in 1983, the first desktop workstation under \$15,000.

In 1984, Apollo created the first 1MIP-class workstations and the first network server. The following year, the company became the first in the

industry to expand network-wide virtual memory over remote links, the first to offer advanced user interface on 32-bit workstations, and the first to offer both the UNIX System V and Berkeley 4.2 versions of the UNIX operating system on a workstation.

Apollo continued this record of technological leadership in 1986, with the first color workstation under \$20,000 and the first open-bus workstation under \$10,000. The company was also the first to integrate personal computer processors into desktop workstations. Apollo also reached two industry milestones in 1986, becoming the first computer manufacturer in the industry to reach the 20,000 workstation shipment and, later, the 30,000 shipment landmarks.

By 1987, Apollo's family of compatible 32-bit UNIX-based computers had expanded to include advanced personal workstations, sophisticated graphics systems, and a wide range of high-performance compute servers, server processors, and printers. Apollo complements its hardware products with software that increases the productivity of technical professionals and offers a full selection of programming languages and a library of high-level support packages for software development, data base management, language testing, and administrative assistance. Apollo addresses the technical work group's need for powerful graphics capabilities with a comprehensive set of unique productivity tools and sophisticated graphics packages based on industry standards.

As Apollo enters the late 1980s, its workstations have become widely recognized as the unifying technology for the many levels of technical and corporate computing. Apollo's Public DOMAIN computing environment transforms the wide range of corporate resources—including personal computers, minicomputers, superminicomputers, supercomputers, mainframes, and special-purpose computers—into a single, powerful computing system. Apollo provides a variety of communications bridges for interconnecting DOMAIN networks via industry standard hardware transports, such as ETHERNET, and for interconnecting DOMAIN networks with other vendors' systems via standard protocols. At Apollo's headquarters in Chelmsford, Massachusetts, more than 1,700 DOMAIN workstations are linked in the largest operating local area network in the world. Apollo's corporate network also includes minicomputers, personal computers, and special-purpose systems from many other vendors.

Apollo workstations are widely used for computer-aided engineering, design and manufacturing, technical publishing, software development, artificial intelligence, life sciences, architectural engineering and construction, and other technical, scientific, and engineering applications.

Apollo's DOMAIN System workstation computing environment is used by many of the world's aerospace, automotive, and electronics leaders, as well as engineers and technical professionals in a broad range of other industries. As of October 1986, the company's customer base had grown to more than 1,700 corporate accounts worldwide. In addition, Apollo enjoys working relationships with many of the world's foremost colleges and universities.

By the end of 1986, Apollo had recorded more than \$900 million in worldwide sales (1981-1986). In 1982, Apollo recorded \$18.1 million in sales, and in the following year sales increased more than 400% to \$80.6 million. Apollo sales nearly tripled in 1984 to \$215.9 million, and in 1985 sales rose to \$295.6 million, despite a widespread computer industry slump. The company's sales continued to increase in 1986, reaching approximately \$392 million.

A worldwide company, Apollo has more than 75 sales and service locations in North America, Europe, and the Far East, including offices in major cities throughout the United States and Canada and in two dozen countries overseas.

Apollo has more than 750,000 square feet of manufacturing capacity in facilities in Exeter, New Hampshire, and Livingston, Scotland. Apollo's worldwide headquarters in Massachusetts also serves as the base for the company's research and development, marketing, sales and service, and finance and administration organizations. International operations are centered in Geneva, Switzerland. Apollo employs more than 3,500 people worldwide (December 1986).

The company is headed by an impressive group of experienced senior industry executives. Apollo's President, Chief Executive Officer, and Chairman of the Board, Thomas A. Vanderslice, came to Apollo in 1984 from GTE Corporation, where he was president and chief operating officer. Other members of the senior management team include Senior Vice President of Finance and Administration and Chief Financial Officer, Richard P. Bond, who joined Apollo after spending more than 30 years at Stauffer Chemical Company, and Senior Vice President of Technology, Marketing, and Manufacturing, Roland D. Pampel, who came to Apollo after holding top positions at AT&T, Prime, and IBM. Two original company founders, J. Michael Greata, Vice President of Research and Development, and David L. Nelson, Ph.D., who serves as Vice President and Chief Technical Officer, are also part of Apollo's senior management team.

DAVID L. NELSON

APPLE COMPUTER, INC.

INTRODUCTION

Apple Computer, Inc., is a leading company in a young and rapidly growing industry—personal computing.

In 1984, the industry was only 7 years old, yet sales of personal computers (ranging in price from \$1,000 to \$12,000) approached \$55 billion. According to InfoCorp, a research firm in California, sales of personal computers from 1985 to 1989 will approach \$238 billion.

Behind these dollar figures lies the one-on-one relationship that has developed between the individual and the personal computer. According to InfoCorp, 12.8 million personal computers were purchased from 1978 through 1984. Estimates are that the number will swell to over 55 million units through 1989.

APPLICATIONS OF APPLE COMPUTERS

Because of their low cost, small size, and ease of use, personal computers are being used for a wide variety of applications. Apple computers are used to predict production costs and profits; create charts and graphs for boardroom presentations; monitor oil-drilling operations; process payroll and inventory for small and medium-size businesses; help students at all levels improve their skills in mathematics, spelling, science, and other subjects; and even to administer and score personality tests. They are used by lawyers in the courtroom and by insurance executives making sales calls.

Apple has helped define the personal computer market by educating people to the potential of these machines and developing systems and software in direct response to customer needs.

Apple was among the first to recognize the importance of convenient product availability and fast, complete service. Today, the company has one of the most extensive personal computer sales and service networks in the world.

APPLE TODAY

Incorporated in 1977, Apple designs, manufactures, and markets personal computers and systems for use in business, education, and the home. By early 1985, approximately 2.9 million Apple systems had been sold worldwide.

Apple has two major product lines: The Apple II family of computers, which includes the Apple IIe and the Apple IIc; the Macintosh family of

computers, which includes the Macintosh 128K, and the Macintosh 512K. In addition, Apple manufactures accessory products for these product lines, which include printers, monitors, modems, etc.

Manufacture of the Apple II family for the U.S. market is undertaken in Carrollton, Texas, whereas Apple's facility in Cork, Ireland, produces these products for the European market. The Macintosh line of products is manufactured in one of the most highly automated manufacturing facilities in the nation, in Fremont, California. Peripherals for both the Apple II and Macintosh lines are also developed and manufactured in a facility located in Garden Grove, California. In addition, Apple maintains a facility in Singapore for the production of logic boards used in their computers.

The Apple II family of products traces its roots to the original Apple computer, designed by Apple cofounder Steve Wozniak. Built around the 6502 microprocessor, these 8-bit computers have become popular because of the enormous amount of software programs written for the machines. The Apple IIc is a transportable computer for consumers and small home businesses. Its applications include personal productivity, education, and entertainment. With its expansion slots and hard disk support, the Apple IIe is designed for businesses and schools.

The Macintosh family includes sophisticated personal computers for businesses, which are known for their ease of use. Powered by MC 68000 32-bit microprocessors, these products feature high-resolution graphics, a mouse pointing device, cut-and-paste integration between applications, and a unique and consistent graphic interface that allows users to improve their productivity without having to learn difficult computer command sequences. Weighing less than 20 pounds, Macintosh delivers exceptional computing power and unprecedented ease of use at an affordable price, creating a price/performance value unmatched in the business market.

The Macintosh 128K and the Macintosh 512K are intended to be used by small and medium-sized businesses, departments of large offices, and college students. The Macintosh Office further positions these computers toward small and medium-sized businesses with the AppleTalk Personal Network and the LaserWriter printer. A low-cost network, AppleTalk offers business users the capability to link up to 32 Macintosh personal computers, allowing users to share high-performance peripherals such as file servers and printers. The LaserWriter offers high-resolution, near typeset quality output and is compatible with the AppleTalk network. Both are easy to install and use.

Many independent companies also make equipment and write programs for Apple computers, assuring users of a wide selection of hardware and software with which to expand their systems. Through Apple's Certified/Registered Developers Department, third-party software and hardware developers can receive the support they need to effectively develop and market their products.

The international market is a large and growing one for Apple. The company began manufacturing personal computers in Cork, Ireland, in 1981 to serve the European market. At the same time, it opened a European marketing and sales headquarters in Paris. Apple now has wholly owned subsidiaries throughout the world, including the United Kingdom, France, Germany, Italy, Belgium, Netherlands, Republic of Ireland, Austria,

Sweden, Australia, Canada, and Japan. In early 1984, Apple formed a joint venture with Mexican investors called Apple de Mexico to manufacture and market the Apple IIe for Mexico and the Latin American countries.

Apple's internal sales organization has grown significantly in the past few years, in response to the changing needs of our distribution channels. In 1984, Bill Campbell, executive vice president, U.S. sales, established a 350-person field sales force to call on Apple's 2,200 U.S. dealers. The sales organization now encompasses a direct sales force that includes value-added resellers and original equipment manufacturers, national accounts, major accounts, education, universities, direct marketing development, and legal. Retail sales include products and programs, channels management, and field marketing and co-op.

THE FIRST APPLE

The history of Apple Computer begins in early 1976 when two young, self-made engineers collaborated on a small computing board for personal use. Steven P. Jobs, then 21, and Stephen G. Wozniak, then 26, took 6 months to design a prototype and 40 hours to build it. They soon had an order for 50 of their personal computers.

With that first order in hand, they raised about \$1,350 by selling a used Volkswagen van and a programmable calculator. They set up shop in Job's garage and soon were doing well enough to form Apple Computer Company, with Jobs as business manager and Wozniak as engineer. They named their computer and the company Apple, because an apple represents the simplicity they were trying to achieve in the design and use of their computers.

That first computer—sold in kit form to electronics hobbyists—was so successful that demand soon outstripped the capacity of Job's garage and overtaxed their capital. Believing they had a product with great commercial and social value, Jobs and Wozniak set out to find professional managers.

Their first recruit was A. C. "Mike" Markkula, Jr., whom they met through a mutual friend. Markkula had successfully managed marketing in two semiconductor companies that had experienced dynamic growth—Intel Corporation and Fairchild Semiconductor.

After researching the personal computer market and assessing Apple Computer's chances, the three men developed plans for acquiring the necessary capital, management expertise, technical innovation, software development, and marketing. Initial financing came from Markkula and a group of venture capitalists that included Venrock Associates and Arthur Rock and Associates. Apple Computer, Inc., was incorporated on January 3, 1977.

Apple remained a private corporation until December 1980, when it made an initial public offering of 4.6 million shares of common stock. In May 1981, there was a secondary offering of 2.6 million shares of common stock by approximately 100 selling stockholders, all of whom acquired their shares through employee stock plans or private placement.

Apple has grown in 8 years from a two-man operation to an international corporation of more than 5,500 employees. Apple sales for fiscal 1984 were \$1.5 billion, a 54% increase over the previous year. In 1983, Apple

became the youngest company to enter the ranks of the Fortune 500 index of U.S. industrial corporations. Apple now ranks number 234 on the Fortune 500 index.

Since April 1983, Apple's growth has been directed by John Sculley, president and chief executive officer. Sculley, former president of Pepsi-Cola Co., guided that beverage company through 5 years of growth that outpaced its industry. Under his leadership, Apple has refined its product line strategy and is beginning the transition to a multiproduct, multibillion dollar company.

BARBARA KRAUSE

APPLICATION SOFTWARE, EVALUATION (see also Applications Software Reviewing)

OVERVIEW

Due to the rapid and significant decrease in the cost of computer hardware, along with the concurrent rise in the availability of software to perform a wide range of specialized functions, a large population of users are faced with the task of selecting and evaluating computer software. This article looks at the general nature of application software selection and evaluation. Topics include sources of software, need for software selection, locating software, the selection process, and the evaluation and comparison of software.

SOURCES OF SOFTWARE

Traditionally, application software, or computer programs that perform one or more user-specified tasks, was custom designed by experienced programmers in-house. Today, application software is available in three additional forms: modular, "canned," and "turnkey." Modular software is a programming tool, such as VisiCalc or DB Master, that enables persons with limited programming skills to develop useful computer programs with a minimum of effort. Canned software is the prewritten, mass-marketed software available from a variety of vendors. A complete package, which includes all the hardware, applications programs, training, and maintenance, is known as a turnkey system. Generally, a turnkey system is bought or leased with ongoing charges for maintenance and software support. Canned and modular software require more participation on the part of the user for implementation.

A variety of commercial sources produce and distribute software. Fifteen to 20 years ago, manufacturers virtually gave software away. At that time, hardware cost exceeded software cost. In comparison, software was cheap, and hardware could not be marketed unless it did something useful. Because IBM unbundled software from computer systems in 1969, hardware manufacturers have become software vendors as well. Manufacturers are acquiring software houses and will contract to write custom-made programs or sell, as general-purpose programs, custom-written programs developed for other users. Even in the personal computer market, IBM has been selling software direct to its largest customers.

Many businesses have found that it is uneconomical to maintain in-house programming staffs. As a result, software houses have become a growth industry. There are over 3,000 software houses at the present time. Some concentrate on contract or custom-designed software, but the trend is toward writing mass-produced, mass-marketed, general application software.

A third commercial source is the software broker who acts as intermediary between the developer and the buyer. The products and services that

brokers offer vary greatly. There are some good packages by designers who lack marketing resources and some mediocre packages designed for one installation and patched for package sale. Some brokers are fully staffed to install, maintain, and support packages, whereas others point to the original developer for these services.

Initially, retail hardware/software stores supported the hobbyist but are now eyeing the lucrative small business market. In the West and Southwest, there is a chain of office equipment/computer stores, and even bookstores are now selling software. The advantage these stores have over other commercial sources is that they are local and, with the exception of bookstores, offer some level of support for the software they sell. In some cases, they can adapt or develop special application packages. Frequently, they offer a saving on a total hardware/software package. The disadvantages are that the software is usually limited to "best sellers" and is rarely under warranty. Also, salespersons often attempt to fit users' needs to the software that the store carries or the software with which they are familiar. This does not always guarantee the best match.

There are also a number of miscellaneous commercial sources. Some large corporations have created spinoff companies in an attempt to recoup investments, or make a profit, from programs developed in-house. However, many of these companies fold as fast as they open because they lack the time, talent, or money to market their product. Software can be bought, without support or warranty, through mail-order houses. Various data bases that serve as software locators will vend software as well. A marketing tool developed by Ocean Data Systems, known as C-DOS, permits the downloading of demonstration programs from a data base for preview before ordering. In the future, disks may be produced and sold on-site using a similar system.

The commercial software scene is currently undergoing rapid and significant changes. Many of the thousands of software companies will go out of business over the next few years. The cottage era of software production is coming to an end. The new entrants in the software market, such as CBS, McGraw-Hill, Dow Jones, and the international accounting firm of Arthur Young and Company, are large corporations that have established marketing and distribution systems.

It is predicted that the largest companies, such as IBM, will concentrate on the Fortune 1000 companies that now account for 24% to 33 % of the market. The small companies that survive the market shake-up will either be writing application software for the big companies or catering to the needs of specialized vertical markets. Vertical market suppliers are better able to target their audience and develop their distribution channels. Furthermore, specialization makes it easier to provide software training and support.

Competition in the marketplace has had a positive impact on application software. Commercial outlets have increased support to maintain and attract customers. Also, producers have become responsive to user criticism. In response to consumer demands, multiapplication software has been introduced to combine several applications into a single package. Currently, these packages combine word processing, electronic spreadsheets, and data base management tools. In addition, designers are attempting to make software easier to use by eliminating the need for extensive control languages and keystroking by introducing innovations such as the mouse and windowed software.

Another new direction for the software industry is the crediting of software authors with the development of a specific package. In the past, software has been identified only by title or distributor. Author identification makes it easier for the consumer to identify not only specific software packages but quality products as well.

In addition to the commercial sources mentioned, there are a number of other ways to acquire software. User groups offer the free exchange of programs developed by their members. Some groups have an extensive bank of programs, as well as national connections. Although most of these center around hobbyists, there are a few that are work or profession oriented. Most hobbyists work for a living, so it is possible to find some application software even within their ranks. The quality of this software is highly variable and seldom screened. Although some will argue that these programs have undergone many revisions and enhancements, most are poorly documented and often difficult to support.

A function that user groups serve is to exert pressure on commercial producers to provide and service quality products. Professional user groups are especially effective in this respect. The banking and insurance industries have written software standards for their respective fields.

Programs can be found in books and periodicals. They do, however, require time to type the lines of code and to learn to operate. The majority of programs found in these sources are geared to the hobbyist or educator, although there are also some general application programs, particularly for small businesses.

Finally, there is the category of public domain software, available through a multitude of sources—educational agencies, hobbyists, software exchanges, information centers, electronic bulletin boards, and dial-up services. Public domain does not necessarily mean free; it simply means not copyrighted. A few software clearinghouses have been established to promote the sharing of ideas and programs. Materials can often be obtained for the cost of a disk. Softswap, operated by the San Mateo Office of Education and Computer Using Educators, and the Apple Avocation Alliance are but two examples of such clearinghouses. The quality and depth of their banks depend on whether they screen materials and the length of time they have been in operation. School districts, universities, and other local educational agencies are sources of public domain software, as many are recipients of federal grants to develop programs and support computing. Lists of projects sponsored by the National Science Foundation and the National Institute for Education can be obtained by writing to these agencies.

NEED FOR SOFTWARE SELECTION

Selection of proper program packages is what makes a computer system an efficient tool. According to Brownstein and Lerner, 50% of data processing software is acquired with selection conducted by personnel with little or no selection experience [1]. Given the fact that there is no such thing as an exact fit of canned software and the complexity of software prevents total debugging, selection should be of concern. Many people buy microcomputers before they have determined their needs or the availability of useful programs to satisfy those needs. The result is a hardware system without the

necessary software to get the job done. Either little software is available or the available software is not compatible with the target machine. A useful approach when selecting a system is to select the software first. There is evidence that this approach is being followed, as the current market in hardware is being software driven.

It is often more advantageous to buy rather than write software. Writing original software is a laborious, time-consuming process. There is a high demand for qualified programmers, particularly in any given application area. The best programmers are working for software houses or computer manufacturers and not for individual companies or low-paying institutions. Commercial software offers the advantage of being ready to use immediately, without the need of in-house programmers and with cost and performance established. Generally, commercial software is better documented than in-house software, which makes support and maintenance easier. Of course, the possible disadvantages of inefficiency, inflexibility, expense, poor support, poor documentation, and lack of acceptance have to be weighed against the in-house production of tailor-made software, which gives the user full control.

Ten years ago there was no software industry. The microcomputer, more than any other factor, has changed that. Today, a \$2,000 microcomputer provides the same capabilities that a \$250,000 mainframe did 10 years ago. The dramatic drop in hardware price has made computing power available to individuals and institutions with limited resources who had never dreamed that computing would be within their reach. Although hardware has decreased in price, the same cannot be said for software, particularly in the newer application areas. Software accounts for 70% to 80% of the cost of a computer system.

With so much software available, selection is an important part of the acquisition process. A survey conducted by Sofsearch International, based on product listings for all types of computers in its data base, showed a minimum of 32,241 packages available in 90,000 operating versions. Broken down, there were 25,509 application packages of which 19,122 were pertinent to business, a profession, or an institution [2]. As a means of comparison, 10,000 application packages were identified in 1982. This shows a growth in 2 years of roughly two and one-half times the number of products. In 1983, software sales reached \$2.2 billion (approximately the size of the record industry) with predictions of \$ 11.7 billion by 1988. And most software sales are for packages under \$10,000 [3].

THE PROBLEM OF LOCATING SOFTWARE

Due to the multitude of application packages available, some searching is required to determine whether there is a package that will meet one's needs. A preliminary report of a computer technology survey in nonprofit organizations shows that 77.8% of the respondents expressed moderate or great difficulty in finding or developing software [4]. Ten years ago, a similar lack of awareness of existing programs led to much duplication of efforts. Practically every installation was writing its own payroll package. This explains why so many payroll packages are available today.

Rather than there being a problem of not enough information, there is a problem of information scatter. Software advertisements are appearing not only in computer magazines but also in professional journals. In addi-

tion, these periodicals frequently review specific software and announce new software as well as products under development. A number of journals and magazines publish annual buyer's guides. Many software advertising brochures are sent, unsolicited, to schools, libraries, and businesses. Computer newsletters, electronic bulletin boards, and popular magazines are also sources of information.

However, only a finite portion of the commercially available software has been captured bibliographically. Bibliographic access can be gained, primarily, through one of three tools: indexes, directories, or evaluative reference/locator services available in both print and on-line format.

There is a great range and variety in the information these tools provide. Indexes generally lead to review articles appearing in the periodical literature. In many indexes, "review" or "evaluation" appears as a sub-heading under the heading "computer programs." However, finding a review for a particular piece of software is difficult. Only a small amount of software has actually been reviewed critically. Directories provide everything from a simple listing of software by name and/or vendor to comprehensive descriptions of the programs.

In the category of reference/locator services are those tools that evaluate or compare similar programs, which are frequently updated. On-line services are updated more often—usually monthly. In addition to descriptive data and review citations, on-line services may provide abstracts of reviews and permit software purchase, usually with discounts of 10% to 40%. On-line services also offer a greater number of entry points. They may be searched by application type, title, or computer brand or operating name, producer, producer's address, memory range, release data, price, or availability of warranty.

A derivative of the on-line locators are the personal software locators. These specially trained consultants search the electronic data bases and provide printouts and applicable reviews for their customers. An annotated list of some locators and other tools used for identifying software is provided in the Appendix. This list is by no means comprehensive but is intended to demonstrate the type and scope of these tools.

In spite of the many print sources available, much of what is considered by users to be good application software is found serendipitously. Word of mouth is the most common way to learn of software. Most published software selection guides recommend speaking with other users in the field and attending conferences as a starting point for locating software.

Although bibliographic control of commercial software is still in its infancy, public domain software is virtually uncontrolled other than by the developers themselves. Tapping into this software source is done almost exclusively through direct contact. Professional organizations, user groups, electronic bulletin boards, and software exchanges are ways of initiating this contact. Because so little software has been reviewed, talking with users in the field is not only a good way to find out what is available but also a way to gather preliminary information for the selection process.

SOFTWARE SELECTION PROCESS

A review of the literature pertinent to software selection and evaluation seems to suggest two basic approaches to the selection of canned software. One could be labeled the marketplace or infusion method. This approach

attempts to determine what software is available and how to fit it to an operation or how to make an operation fit the software. This was the approach taken, if not recommended, when relatively little software was available. Users were often relieved or amazed to see that something "worked" and did not question the quality. The result was the fostering of many poor products.

Partially in response to the many mediocre products, a more analytical approach is now recommended. Training for implementing this approach is now under way. Many guides and articles are being written for specific and general audiences on the selection of prewritten software. Perhaps, most significantly, these guides are directed to the user rather than to data processing personnel.

The first step in the process is to perform a needs assessment by taking a careful look at how things are being done, the flow of the work, and the objectives. While this information is being gathered, there is a question to keep in mind: Is automation appropriate? In about one quarter of the cases, the manual system is found to be efficient, or errors are found and corrected to make the system efficient. Because of the aura surrounding technology, it is often assumed that automation is superior to other methods. As maturity and experience with software develops, some of the mystique that makes it appealing is dispelled. On some occasions, more relevance is achieved by other means. If automation is the answer, the self-analysis is still important for identifying and correcting errors before incorporating them into the automated process.

After the decision to automate and buy software has been made, it must be determined what the software is to do and how it is to do it. Consideration should be given to which functions are mandatory and which are "nice to have." At this stage, software requirements and specifications should be written. Because the software may be in use for a long time, thought should be given to future demands and functions. When establishing minimums for performance, a good rule of thumb is to use present minimums plus 25%.

Identifying potential packages and gathering preliminary information is the next step in the process. Sources for identifying and locating software have already been discussed. The type of information accumulated at this stage should include name, vendor, hardware and system requirements, overview of functions and features, capacity, and price.

Packages that clearly fail to meet the objectives and specifications established earlier can be eliminated. It will be obvious if the hardware requirements are different from those available or are too costly to obtain. If objectives are not met, the package should be eliminated rather than altering the objectives to suit the package. With the amount of application software that is being produced and improved, there is little reason to settle for a package that offers less than is desired. Some waiting or additional searching may be required.

After the field has been narrowed by the selection process, a detailed evaluation and comparison of the remaining packages takes place. This is discussed in more detail in the next section.

The evaluation should lead to a decision and the negotiation and/or purchase of the appropriate software package. If an expensive package, one that requires modification, or custom software is being purchased, the vendor's responsibilities and software performance should be committed to

a written contract. Typically, the vendor will try to assume minimum responsibilities. For this reason, standard contracts are discouraged. What is to be done, by whom, how, when, and at what cost is important, especially if the vendor is providing the majority of training, support, and maintenance.

EVALUATION AND COMPARISON OF SOFTWARE

In the past, software evaluation was generally considered to be the domain of the data processing department. However, many of today's potential software buyers are not affiliated with a data processing department. Furthermore, a programmer's opinion of what constitutes good application software is not necessarily the same as that of the end user. On the other hand, a novice may view even the slowest, most inflexible software as a remarkable improvement of a manual system.

If there is a unique problem that will require a considerable amount of special or expensive software and there is no time or personnel to conduct an analysis of present conditions, identification of software, and comparison of packages, an outside consultant should be employed. Such a consultant should have expertise in the application area and no vested interest in any products under consideration. It is therefore questionable to use a software or hardware salesperson as a consultant. The temptation exists to fit the application to the available product rather than match the software with the application. Knowing hardware and software is not enough for choosing application software. It is important that a consultant have a full appreciation of the intricacies of the application and the user's objectives.

For many, there is no decision to make concerning the hiring of an outside consultant. It is simply outside the budget, or the cost of the software itself does not warrant this type of expenditure. The end user, however, is not a bad choice for evaluator. Who other than the user has a better understanding of present conditions and desired objectives and requirements? Those who train users to evaluate software agree that by using subject expertise and knowledge, users can become effective evaluators. Confidence in the user's own expertise must be bolstered. Basically, this requires the demystification of software through knowledge and experience. The process of identifying software and gathering information itself can provide a great deal of background.

The person or persons selected to do software evaluation should possess three characteristics: (a) expertise in the application area, (b) general knowledge of any existing hardware system, and (c) no vested interest in the performance or nonperformance of a software package. If the end user is not directly involved with the evaluation, he/she should at least provide some liaison. This will be beneficial not only in choosing the most appropriate software but also in implementation later. The make and model, language, operating system, capacity, and peripherals constitute the required hardware knowledge. Fear of automation or an attitude known as the "not invented here syndrome" can interfere with objective evaluation. In simple terms, use the expertise at hand to best advantage when choosing evaluators.

In order to evaluate software, there is a need to establish some points of reference or criteria. Unlike other purchases, software does not stand

alone. Criteria can be categorized as being application specific or nonapplication specific. The application-specific criteria are the features that meet the objectives and requirements identified during the selection process. The nonapplication-specific criteria fall into three broad areas: (a) technical and operational, which deal with the design concepts, ease of use, and hardware compatibility; (b) implementation and maintenance, which deal with documentation, training, modification, vendor support, and vendor reliability; and (c) price, which includes the direct purchase or lease costs, as well as the indirect costs such as training, modification, enhancement, conversion, and operation.

Much has been written concerning the need for establishing criteria, and numerous checklists can easily be located. This in itself creates difficulty. Rather than simplifying the process, elaborate evaluation forms run the risk of creating confusion. They have become so cumbersome that one can easily lose sight of the overall performance of a software package. Another difficulty is that most checklists are general in nature and tend to foster a mechanical approach to viewing software.

Educators who, as application users, have had considerable practice in evaluating software during the last few years have begun to question what constitutes valid evaluation criteria. The idea that not all criteria are created equal is emerging. Recent forms attempt to assign weights or ranks to the criteria. However, numbers should not be relied on to convey important substantive matters. What is needed is a way to distinguish between deficiencies that may be easily remedied and those that cannot be remedied without overall reconceptualization. In the end, evaluation of software, as with most performance items, is subjective. Although a checklist is convenient for comparing similar programs, it should be used as an informative rather than authoritative tool.

The user must develop confidence in his/her own ability to evaluate software. This requires some knowledge of what is available and how it works. By keeping abreast of the current literature in one's own field, occasionally browsing through popular computer magazines, and talking with software vendors and users, one can develop an appreciation for what is available. Reading software reviews, as part of the identification and information-gathering step, helps the potential user to formulate the criteria that he/she will use to evaluate and compare software packages.

A skeletal checklist can provide the user with a format for identifying and prioritizing application-specific criteria without prejudicing viewpoint. In such a checklist, criteria can be divided into three parts: (a) application suitability; (b) controls and security, which should provide for backup or copy in addition to restricting access; and (c) operational performance. Once all criteria have been determined, they can be ranked by the user with the highest number assigned to the most important characteristic. As a particular package is evaluated, the evaluator can assign a number between 0 and 10 for each characteristic. By multiplying the characteristic's value by the assigned rating, the evaluator can produce a numerical score comparing similar software packages.

For personal, or "buy, don't buy" decisions, a four-section criteria list identifying application features, technical and operating requirements, implementation and maintenance requirements, and price range should be sufficient. It is possible to include both mandatory and "nice to have"

TABLE 1 Establishing Evaluative Criteria

APPLICATION: Grade book

DESIRED FEATURES

Grade computations for up to 35 students/5 classes
 Accepts at least 25 scores per student
 Teacher-defined formula for grade computation
 Searchable by name, class, or subject
 Grade report to include name, grade level, subject,
 class, total points, average, and letter grade
 Progress report generator
 Roster printouts
 Attendance recording option

TECHNICAL AND OPERATING DATA REQUIREMENTS

TRS-80 model III, one disk drive

On disk

48K of memory or less

Backup

Editing

Error trapping and diagnostics

Source language-BASIC

IMPLEMENTATION AND MAINTENANCE REQUIREMENTS

Available immediately

Documentation to include detailed instruction manual—
 no other training required

Access to updates and enhancements

Local support or hot line number

PRICE

\$60 or less

features by marking mandatory items with an asterisk or other distinguishing mark. A sample of such a checklist is provided in Table 1.

Actual use of a software package on the user's own equipment, with real data and situations, is the best means for evaluating software against established criteria. The evaluator should also consider the ease or difficulty of interaction, whether data are checked for validity before processing, and whether the screen displays are appropriate. Some vendors, particularly in the educational field, will permit a 30-day preview period. If the vendor does not permit preview, one should attempt to arrange a benchmark test on the vendor's equipment. Most local retail vendors have some type of demonstration facilities. If remoteness or need for a large file conversion makes a benchmark test impractical, the vendor should be asked to identify nearby installations where the package can be observed in operation under conditions as close as possible to those of the intended user. Arrange for a visit and prepare specific questions based on the criteria list. Be sure to inquire about any difficulties or unmet claims with either the package or the vendor.

If none of these previous methods is available, the vendor might conduct a partial demonstration of the software or provide a special demonstration version of the program. Caution should be exercised in these circumstances. Often, only the best and most impressive features are shown while faults are suppressed.

Reviews are better used in deciding whether to preview a piece of software rather than in place of preview. Reviews can become obsolete because software can be quickly and easily changed. Also, the version reviewed may be different from the one under consideration. Expertise and background of the reviewer is another question. Many reviews are unsigned and unsolicited. Some magazines do not publish negative reviews. Even if a review is favorable, the reviewer may have used different objectives, worked under different conditions, or looked only at the diskette and not at the entire system. Software should not be purchased sight unseen.

While previewing software, look at the accompanying documentation and vendor's service and reliability. Is the documentation provided written in clear, concise English? Is sufficient information provided for training and operation? Is technical information, needed for repair and maintenance, included in addition to tutorials and operating instructions? Will source codes be provided or obtainable should the dealer not be available for support? The vendor's stability, reliability, and expertise of personnel should be investigated, especially if the vendor is to provide training and maintenance or a custom product. Determine the provision, the time, and the cost of services.

With so many canned programs available, it is probably worth the time and effort to search for a software package rather than attempt to write application software in-house. Plenty of information and resources concerning software selection and evaluation exist, and the user, with some background, can become an effective evaluator. The most crucial step is to carefully identify the objectives and specifications that the software is to meet. It is doubtful that any one software package will meet all requirements exactly. The key questions are how much adaptation is required and is it acceptable? The final consideration is the availability and extent of support. The specific method applied to the selection and evaluation of software, as well as its thoroughness, will depend on the available resources—finances, personnel, and time.

REFERENCES

1. Irvin Brownstein and Nancy B. Lerner, *Guidelines for Evaluating and Selecting Software Packages*, Elsevier, NY, 1982, pp. 1.
2. "News and Notes," *Library Software Review*, 3, 18 (March 1984).
3. Andrew Pollack, "Slugging It Out on the Software Front," *The New York Times*, October 16, 1983, Sec. 3, p.1.
4. "1983 Annual Survey of Computer Technology in Nonprofit Organizations," *Library Software Review*, 3, 14 (March 1984).

APPENDIX: SOURCES FOR IDENTIFICATION OF SOFTWARE

Indexes

Computer Science Resources: A Guide to Professional Literature. Contains references to a number of resources that address the subject of software.

Computer Literature Index. Covers fairly technical magazines and journals in the fields of computers and data processing.

Consumer Index to Product Evaluation and Information Sources.

The Digest of Software Reviews: Education.

Microcomputer Index. Bimonthly coverage of articles and reviews from almost 50 microcomputer periodicals. Includes a subject index whose entries refer to abstracts appearing in the second section.

Small Computer Program Index. Bimonthly guide to listings in personal and small computer magazines and books in the United States and Great Britain (contains printed programs).

Directories

General

Apple II Blue Book. Where-to-find-it guide and descriptions of over 5,000 programs.

Data Sources. Quarterly guide to hardware, data communications, software, and services.

Freeloader 500 Software Library.

IBM PC Software Guide. Contains detailed information on over 3,000 PC and XT software programs from 1,100 suppliers.

ICP Directory. Multivolume annual guide with over 8,000 packages for types and size of computers (no evaluations).

International Directory of Software. Indexes products by category; includes description of software and suppliers profile.

International Software Directory. Available in two volumes: microcomputers and minicomputers. Based on the International Software Database.

On-Line Micro Software Guide and Directory. Directory of product descriptions.

Small Systems Software and Services Sourcebook.

The Software Catalog. Contains extensive data on 25,000 microcomputer packages.

The Software Directory. Includes brief descriptions of software.

Sourcebook. Directory of small systems software and services.

Skarbet Software Directory. For Apple computers.

(continued)

APPENDIX (Continued)

Educational

Educational Software Directory. Lists titles of more than 500 programs for Apple, Pet, and TRS-80 (no evaluations).

The Educational Software Selector (TESS). Contains comprehensive and detailed information on the availability of educational software for microcomputers for nursery school through graduate school (citations to reviews).

Reference Manual for Instructional Use of Microcomputers. Cross-references over 1,000 educational software programs. Indexes journals and books and contains some evaluations.

School Microware: A Directory of Education Software. Description of software for classroom use with Apple, TRS-80, Pet, and Atari. Brief descriptions are organized by grade level, subject area, and hardware system (no evaluations).

Swift's Educational Software Directory. Directory of educational software.

Library-Specific

Directory of Information Management Software. Provides basic product information on commercially available packages for libraries and information and record centers.

Library and Information Sciences: A Directory of Computer Software Applications.

*Reference/Locator Services**Print*

Application Software Reports. Limited number of packages described.

Computer Software

Datapro Directory of Software. Over 2,000 packages, indexed by vendor and product; includes information on function, hardware, and peripherals required and price, maintenance, documentation, and training.

Datapro Directory of Microcomputer Software. Over 2,000 packages for microcomputers; includes vendor profiles, hard-to-locate supplies, features on software and basic product information; identifies best packages according to user ratings.

Datapro Reports. Compares vendors claims against users' experiences.

Micro-Courseware PRO/FILES. In-depth evaluation of educational courseware by EPIE/Consumers Union.

MicroSIFT Courseware Reviews. Educators' guide to evaluated, field-tested software.

Micro Software Report: Library Edition. Focuses on software programs designed for the microcomputer; mainframes and minis included if interface with micro available.

(continued)

APPENDIX (Continued)

Software Reports. Updated, loose-leaf reference guide to evaluated educational software. Stated criteria used to evaluate courseware in 20 subject areas.

On-Line

C-DOS (Ocean Data Systems). Permits search of data base and demonstration of software at computer stores.

The Computer Database (DIALOG File 275).

DISC (BRS). Cover-to-cover indexing of journals in microcomputing with peripheral coverage of minis, EDP, and information science.

FileSoft (BRS/Search and BRS/After Dark). An expansion of On-line Micro-Software Guide and Directory. Contains information on over 2,000 microcomputer packages exclusive of games and entertainment.

The International Microcomputer Software Database (DIALOG File 232). World's largest software data base updated monthly.

MENU (The International Microcomputer Software Database). A personal software locator updated daily with printouts sent via Federal Express on the day of search. Included with printouts are any available reviews. Discount on purchase of software ranges from 10% to 25%.

Micromputer Index (DIALOG File 233). Indexes over 25 English language microcomputer periodicals cover to cover in addition to other publications.

ON-LINE SOFTWARE LIBRARY (Searchmart's Free Access Library). Menu-driven searches. The data base contains about 2,000 packages for mainframes, minicomputers, and microcomputers. Descriptive information may or may not include manufacturer's comments.

One Point Electronic Catalog (ITM). Menu driven with browse or search options. Brief printout includes subject and performance ratings from independent evaluators. Product description includes highlights of key functions, review abstracts, and user comments. Discount, 15% to 40%.

RICE (BRS). A data base comprised of information on the state of the art in educational computer applications.

Sofsearch International, Inc. A personal software locator offering 15% to 40% discounts on over 40,000 software packages. Does not include product description or reviews.

The Software Catalog (DIALOG File 232). Contains information of 50,000 software products for minicomputers and microcomputers. Primarily used as a marketing tool.

GLOSSARY

- Analyst.** A person skilled in the definition and the development of techniques for solving problems, especially techniques for solutions on a computer.
- Application Software.** Programs designed and written to solve specific user-oriented problems.
- Benchmark Problem.** A problem used to evaluate the performance of computer and/or software, relative to each other.
- Data Base.** An organized grouping of data elements.
- Debugging.** The process of identifying and correcting errors in programs.
- Documentation.** All paperwork, manuals, and documents used to maintain a complete record of a system's design, user responsibilities, data entry, and operating instructions.
- Evaluation.** The process of studying the characteristics and performances of software packages to determine the relative merits of each.
- Field.** A set of one or more characters treated as a unit of data.
- File.** An organized collection or records directed toward some purpose; for example, a file of inventory records.
- Maintenance.** Maintaining software by making repairs, updates, and enhancements.
- Modular.** A programming tool that enables persons with limited programming skills to develop useful computer programs with a minimum of effort. Examples are VisiCalc and DB Master.
- Object Language.** Machine language.
- Operating System.** Collection of computer programs for accomplishing such housekeeping functions as input/output control, memory allocation, and program read-in.
- Software Package.** Data, programs, and assistance provided by a vendor, which may be partially modified for the configuration of the user's computer system.
- Source Language.** Language, other than machine language, in which a program is originally written.
- System Software.** Programs designed around the internal environment of the computer to directly or indirectly support processing.
- Throughput.** Total useful information processed or communicated.
- Turnkey System.** Computer, peripheral equipment, software, documentation, supplies, and installation supplied by the manufacturer to provide a total hardware/software system. Generally geared to a particular function or application.
- Unbundling.** Pricing strategy in which the services, programs, training, and so forth, are sold independently of the computer hardware by the computer hardware manufacturer.
- User Group.** An organization composed of users of software packages.
- Utility Program.** Standard program for performing frequently required system tasks, such as sorting, merging, selecting, editing, copying, or printing out the contents of a file.
- Vertical Market Software.** Specialized software designed for limited markets.

APPLICATIONS SOFTWARE REVIEWING (see also Application Software, Evaluation) (ion)

Software reviewing may be characterized as a collection of techniques for the analysis of software by humans to raise issues about any aspect of that software while leaving the resolution of those issues as a separate process. The practice of software reviewing is directed toward the evaluation and enhancement of every artifact related to contemporary software development and use [1, 2]. Group processes are widely used because of concern for complex detail and varied points of view. Figure 1 illustrates a typical interaction of agents, processes, and data flow.

A software review may be called from within the software development process by the author or person responsible for any element of that software. (Software modification is considered to be a special case of software development.) It is generally better if the review process takes place outside of the project development group in order to minimize reviewer blindness and maximize the benefit of multiple viewpoints in the review [3]. The technical review details illustrated in Figure 1 generally proceed as follows:

1. The software author obtains the commitment of someone who is not directly involved in the project to act as review coordinator. The documents that are to be reviewed and specific review goals are given to the coordinator, who assumes responsibility for the completeness of the review materials. The amount of material to be reviewed should be guided by what a group review meeting can accomplish in 30 minutes to 2 hours [cf. guidelines in Ref. 2].
2. The coordinator schedules a review meeting with a group of peer reviewers who appear to have appropriate expertise and complementary viewpoints relative to the goals of the review. Materials are distributed, and the reviewers are given 3 to 10 days [1] to prepare for a meeting. During this time they each review the documentation individually. Review guidelines and checklists, together with the reviewer's specially selected expertise, guide this process.
3. At the meeting, the coordinator appoints a recorder and assumes leadership of the meeting. The individual reactions of the reviewers are communicated to the group, combined, filtered, amended, extended, and summarized. Issues are recorded carefully, but their resolution is left to the software development group or project management.
4. The issues raised in the review meeting are reported to the software author for resolution. Issues that arise outside the area of responsibility of the software author are communicated to project management in a related issues report. In addition, an evaluation of the software is agreed upon by the review group and communicated to both project management and the software author. This report simply states that the software is, in the best judgement of the review group, either acceptable as is, in need of minor revisions, or in need of major revision and a complete new review.

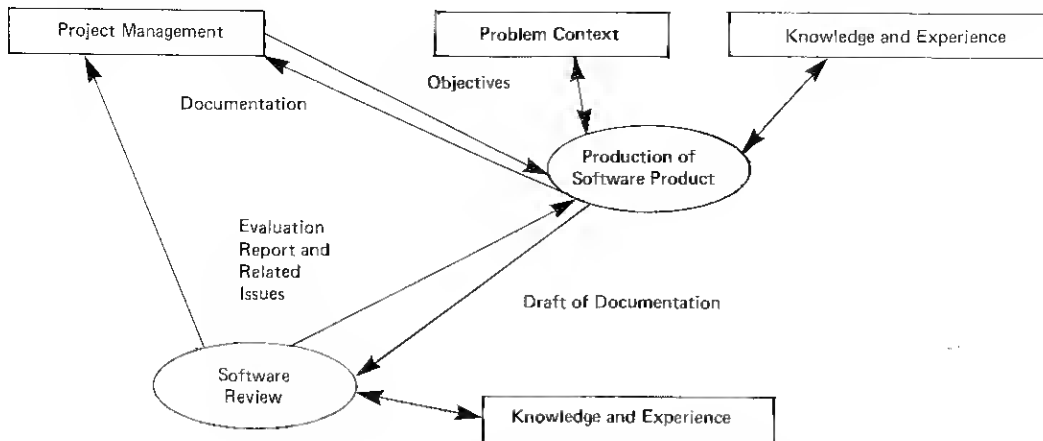


FIGURE 1 Typical details in the technical review of software.

A number of detailed concerns and variations in this procedure are described in Refs. 1 and 4. For example, the popular walkthrough technique [5] consists of a presentation in which the review leader describes the response of the software to hypothetical scenarios in its domain of concern. This approach allows a software element to be shown to many individuals with considerably less preparation than a technical inspection. Walkthroughs are effective at providing group communication, education, and a rough clue to product quality.

The complexity of these procedures and the specific skills required for the various roles demand specialized training. (Fagan [2] asserts that 1 to 3 days are adequate in general.) Guidelines and checklists for reviewers provide a formal way for a development group to structure reviews and learn from experience.

The particular software element that is to be reviewed may consist of a variety of things, such as a specification, a design, a code, or a test plan. It may be custom written, or something that is proposed for reuse or purchase.

The essence of a software element must ultimately be captured in a documented form that does not rely in any way on the involvement of its author in order to be used. At a minimum, this software documentation must provide an organized presentation of the outcome of a software development task in a form that can be verified by comparison with the task goals and validated by its subsequent effectiveness. It must present a clear, complete communication of system concepts, in addition to being internally consistent and unambiguous. These criteria for software quality imply that both technical adequacy and the properties of the software documentation as a written communication must be considered by reviewers.

Current software reviewing practice involves substantial investments in software reviewing [6]. Shelly and Cashman [7] claim that application

programmers spend as much as 25% of their time in software review activities. People are making these investments because they feel that they are gaining a variety of benefits. Software that has been reviewed should require less modification and have fewer defects. Freedman and Weinberg [1] emphasize the value of productivity improvements due to early detection and removal of defects and the measure of software quality that reviewing provides.

The effect of software reviewing on the people involved is also significant. The author of the software is putting his or her work up for critical public view, which provides added incentive to produce a polished product. The reviewers and the author must interact in a task-oriented "egoless" fashion [8] which promotes helpful group dynamics [9]. Finally, reviewers increase their expertise as a result of getting to know the work of their peers [5].

This last benefit from software reviewing—call it education—is an especially interesting side effect for software engineering educators. Lemos [10] and Shelly and Cashman [7] document specific benefits from review techniques in university classes that emphasize programming.

Software developers and educators currently believe that software reviewing benefits the process, the product, and the people involved with software development. However, actual use of software reviewing techniques appears to be uneven, in that some organizations make heavy use of these techniques throughout the software life cycle, whereas other organizations use them mainly in earlier phases of software development. This uneven use of software reviewing, the magnitude of current investments and expectations for reviewing, and the lack of reliable validation of its claimed effects all point to a need for research.

The clear desirability of reviewing a document before using it may seem to be too obvious to require validation. It only makes sense to obtain an objective evaluation of a software element before it is used. There are two major complications with simply accepting the appropriateness of current procedures and the validity of benefits that are attributed to software reviewing. The first is that the practice of software development is rooted in the "soft" character of software. The lack of concreteness in software masks its complexity. Increases in the number of components and interdependencies of software have a profound effect on development cost, schedule, and quality. The net effect is that even a highly skilled software professional is likely to make significant errors in developing and documenting software, which mandates the use of effective defect removal procedures. Such complexity seems to at least call for a detailed understanding of the mechanisms and benefits of a variety of quality assurance techniques.

The second factor that complicates the development and use of valid theory for software reviewing is the popular perception of software as some form of code. This limited understanding of the nature of software gives rise to a reliance on testing to demonstrate its usability and correctness. Unfortunately, many important issues in the specification, design, documentation, and validation of software systems can be overlooked when testing is the focus of quality assurance. Task requirements and specifications cannot be tested until they are in some sort of executable form. In addition, testing alone is not even a dependable measure of correctness. Due to the complexity of software, testing can only demonstrate the presence of errors, not their absence.

On the surface, it appears that software quality assurance and the achievement of communication (or learning) goals define the desired outcomes from software reviewing. Ultimately, concern for obtaining cost-

effective benefits from software reviewing must address software quality, cost, development constraints, and the effect of reviewing on people.

Quality software must be verifiably correct, usable, modifiable, and efficient. Records of subsequent software use, modification, and error detection provide basic indicators of quality, but reliable measures of the intrinsic merits of a software element are generally not available. (Human review processes currently provide the best measure of intrinsic software quality [3].) Software reviewing is often relied on to provide timely defect detection, because the cost of removing a defect is known to grow exponentially with the amount of effort that is invested subsequent to the insertion of the defect [11].

These goals appear to imply that one appropriate measure for the effectiveness of software reviewing should be the number and significance of software defects that are detected and subsequently resolved by software reviewing. However, part of the defect removal effect of a software review may actually occur as a result of the software author's preparation for the review. Thus, it is more meaningful to consider the defects remaining in a software element as a measure of the effectiveness of a defect removal procedure. Additional measures should address issues of development schedule and cost and effects on the knowledge and feelings of everyone involved with the process.

REFERENCES

1. D. Freedman, and G. Weinberg, *Reviews, Walkthroughs, and Inspections*, Little, Brown and Company, Boston, MA, 1982.
2. M. E. Fagan, "Advances in Software Inspections," *IEEE Trans. Software Eng.*, SE-12(7), 744-751 (July 1986).
3. G. M. Weinberg, and D. P. Freedman, "Reviews, Walkthroughs, and Inspections," *IEEE Trans. Software Eng.*, SE-10(1), 68-72 (January 1984).
4. IEEE Software Review and Audit Working Group, "Draft Standard for Software Reviews and Audits (P1028)," prepared by Subcommittee on Software Engineering Standards, IEEE, New York, 1985.
5. Ed Yourdon, *Structured Walkthroughs*, 3d ed., Yourdon Press, New York, 1985.
6. L. Beck, and T. Perkins, "A Survey of Software Engineering Practice: Tools, Methods, and Results." *IEEE Trans. Software Eng.* SE-9(5), 541-561 (September 1983).
7. G. Shelly and T. Cashman, "Implementation of Structured Walkthroughs in the Classroom," in *Review, Walkthroughs, and Inspections*, Little Brown and Company, Boston, MA, 1982, pp. 425-434.
8. G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand-Reinhold, New York, 1971.
9. P. C. Semprevivo, *Teams in Information Systems Development*, Yourdon Press, New York, 1980.
10. R. S. Lemos, "An Implementation of Structured Walkthroughs in Teaching COBOL Programming," *Comm. ACM*, 22(6), 335-340 (June 1979).
11. B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

APPLIED DIGITAL DATA SYSTEMS INC.

BACKGROUND

Founded in 1970 by William J. Catacosinos, Applied Digital Data Systems Incorporated (ADDS), is a wholly owned subsidiary of NCR Corporation of Dayton Ohio.

ADDS began its operation as a manufacturer of video display teletype-compatible terminals, and by 1977 became the leading independent supplier of this product in the United States. The company has since added microcomputers and minicomputers to its product line, and today is known worldwide as a manufacturer of small- to medium-sized business systems.

THE SYSTEM 50, 70, AND 75 PRODUCT LINES

Tracing ADDS' line of computers begins in 1976, when company officials decided to increase the firm's strength by entering the systems arena. The company offered a microprocessor-based microcomputer called the System 70, which ran under ADDS*PLUS, a screen application system primarily used for data entry purposes. The system was housed in a foam-molded desk with a cavity for the terminal. This approach in packaging was considered a main selling point.

Eventually, ADDS*PLUS was replaced by another operating system that supported both BASIC and FORTRAN languages. It was named the ADDS Disk Operating System (ADOS). The company had a closed alliance with Bill Gates, founder of the Microsoft Corporation, and through this operating system became one of the first of his customers to use CP/M Microsoft BASIC. ADDS asked Mr. Gates to put MS-BASIC, which had its own built-in operating system, onto CP/M. Changes were then made to the CP/M to fit it to the System 70, and the end product became the new OS.

Soon after, the System 75 was created, utilizing the latest access-method technology, indexed sequential access method (ISAM). Through the use of MIXSAM, a Peachtree software product that ADDS licensed and integrated into the ADDS BASIC package, the System 75 provided intrinsic ISAM functions.

In 1978, ADDS began designing applications and became an original equipment manufacturer (OEM) of system products, supplying companies such as Radio Shack with the System 50. This product was simply an offshoot of the System 70, which Radio Shack named Tandy 10. In the meantime, ADDS' mainstay continued to be the sale of terminals and systems to dealers.

THE MULTIVISION PRODUCT LINE

As demand for systems products grew, company officials restructured ADDS into two separate product divisions: display terminals and business systems. Management concluded that it could better address the future needs of these separate marketplaces by having each division control its own sales, marketing, and engineering staffs. In 1979, the new Business Systems Division announced a line of small 8-bit computers under the trade name Multivision. Development was extensive, and the first units were shipped in 1980.

Multivision was the first multiuser system of its kind to appear in the marketplace, and it was with this product that ADDS received recognition for being the first company to use Shugart 5 and 10Mb disks.

The Multivision system, which was designed for desktop use, was modular and consisted of three parts, which could be purchased separately and stacked on top of one another. The Multivision system had a CP/M compatible, multiuser operating system (MUON), which supported a broad range of software packages, such as the ADDS extended Microsoft BASIC interpreter, compiler, and a proprietary word processing package.

The basic module of the Multivision computer, which when stacked served as the "middle" module, was marketed as Multivision 1. The unit was smaller than a typewriter and contained a 5MHz 8085A-2 processor. All peripheral interfacing was done via I/O commands, which kept the system's 64Kb of RAM free for programming. Two quad-capacity minidisk drives provided 700Kb of storage. Access time to this data was minimized by a high-speed band actuator with a step rate of 6 msec/track (which was six times faster than some minidisks of the time.) Direct memory access (DMA) capabilities provided memory-to-memory data transfer, memory-to-minidisk transfer and, with the expansion of Multivision 2, memory-to-hard disk transfer. The system had two parallel and two serial printer ports and utilized 256 bytes of nonvolatile memory (which was used instead of mechanical switches). Also included were eight interrupt vectors, three interval timers, and power-up diagnostics using a shadow ROM. Multivision 1 was priced at under \$4000.

The top and bottom module of the Multivision were marketed as Multivision 2. Using Winchester technology, the top module provided a choice of 5 or 10Mb of hard disk storage at a cost of thousands of dollars less than other hard-disk systems available at the time. The bottom module contained memory expansion for multiuser processing.

Multivision 3 consisted of the entire stack and included a module that contained up to 256Kb of RAM and provided a multiuser system that supported up to four display terminals simultaneously.

The main selling feature of Multivision hardware and software was "upward compatibility." Expansion required no change in equipment, no costly conversion, no change to programming languages or utilities, and the same operating system could be used.

Ultimately, Multivision turned out to be a highly marketable little business machine that could be used for everything from cash flow planning and budget preparation to manufacturing planning and sales forecasting. The success of the Multivision system can be judged by the consistent requests dealers have made for it; orders arrived long after ADDS ceased marketing the system in favor of another product line. Shipments continued until 1985.

THE MENTOR PRODUCT LINE AND THE PICK OPERATING SYSTEM

In a new effort to capture the relatively open market of small- to medium-sized businesses that needed affordable and efficient computers, ADDS purchased the license to the Pick Operating System from Pick Computer Works. While marketing the Multivision systems, ADDS conceived the Mentor product line.

By June of 1980, a prototype of a Mentor computer had been created, and by October, ADDS became the first company to port the Pick OS to a Zilog Z8000 (16-bit) microprocessor. The rationale for choosing the Z8000 was that its bit-handling capabilities with the Pick OS made the system 20% faster than competitive miniprocessor systems.

During the development of the Mentor product, there were other significant accomplishments as well. ADDS was the first to use low-cost Priam 14-inch Winchester drives, and the company became the first major OEM account for that drive. Also to its merit, ADDS developed a battery backup system in which battery power is used to power the electronics and disk drive during a power failure. The duration of the battery power is long enough to allow the operating system to transfer system status and the contents of memory to disk before the system shuts down.

Mentor systems are high-performance, stand-alone, multiuser business computers, versatile enough to be used in many general-purpose environments. Several models have been organized around a Zilog Z8001 central processing unit with a 16-bit data bus and a 20-bit address bus, providing ease of expansion from the basic configuration to a more advanced system.

There have been essentially four types of Mentor systems based on this technology: (a) Mentor 3000; (b) Mentor 4000; (c) Mentor 5500; and (d) Mentor 2500.

The configuration for the Mentor 3000, which is no longer marketed, consisted of a 4MHz microprocessor, 256 to 512Kb of RAM with parity, a $\frac{1}{4}$ -inch streaming cartridge tape drive, one to eight multiterminal controllers (MTCs) capable of communicating with eight terminals each, and one or two Winchester technology disk drives with an unformatted capacity of 34Mb. An uninterruptible power supply (UPS) provided power to the disk drive, tape drive, and all of the circuit boards in the Mentor card cage.

The configuration for the Mentor 4000 consists of a 6MHz microprocessor, at least 512Kb of RAM with parity (expandable to 1Mb), a $\frac{1}{4}$ -inch reel-to-reel streaming tape drive, two to eight MTCs capable of communications with eight terminals each, and one or two 150Mb disk drives.

Like the other Mentor computers, the Mentor 5500 system is organized around a Zilog Z8001B central processing unit, but with a 16-bit data bus and a 23-bit address bus. Each system contains a high-speed central processor board, one to three 512Kb universal RAM boards, one to four 8-inch hard disk drives, a $\frac{1}{4}$ -inch reel-to-reel streaming tape drive, and one or two intelligent terminal controller board sets capable of communicating with up to a total of 64 terminals and four high-speed Centronics-compatible parallel printer interface ports.

A physically smaller unit is the Mentor 2500 computer. This system is similar in technology, performance, and purpose to the other Mentor systems. As with the much larger Mentor 5500, the Mentor 2500 is organized around a Zilog Z8001B central processing unit with a 16-bit data bus and a 23-bit address bus. Expansion from the basic configuration to a more advanced system is a simple procedure.

The basic Mentor 2500 system contains an enhanced processor and interface controller board, 256Kb of RAM, a 5 $\frac{1}{4}$ -inch disk drive, a $\frac{1}{4}$ -inch streaming cartridge tape drive, and a printer and terminal serial interface (PATSI) board capable of communicating with up to 16 terminals and one high-speed, Centronics-compatible parallel printer port. The formatted capacity of the disk ranges from 14.7 to 66.7Mb.

Options include a second PATSI board, which provides support for 16 more terminals (the maximum is 32) and a second parallel printer port. Additional RAM of up to 1Mb is available, and a second disk drive may also be added.

Although the hardware technology in Mentor systems has done much to sell the product, the ADDS-enhanced version of the Pick Operating System (the Mentor OS) has added an immense value to the marketability of the computer.

Besides being extremely portable, the Mentor multiuser OS has proved to be an ideal choice for business environments today. Its main feature is a built-in data base management system. Using a simple inquiry language called Info/Access, users with little previous training can type in "English-like" sentences and access data from files and manipulate print formats for reports. Programmers have experienced the benefits of the system as well. The programming language is DATA/BASIC, a sophisticated and significantly enhanced form of BASIC that is well-suited for work with the Mentor OS. A terminal control language (TCL) is also available for accomplishing routines where DATA/BASIC code may be too cumbersome. Terminal control commands may be stored in a file and used in much the same way as standard job control language (JCL).

THE NCR PURCHASE AND THE PC AND MICROCOMPUTER PRODUCT LINE

The next milestone in the history of ADDS appeared in the Fall of 1980, when NCR purchased a controlling percentage of ADDS stock. After the purchase, ADDS was permitted to continue operating in an independent fashion. Part of NCR's business philosophy concentrates on decentralization and promotes free competition by its subsidiaries. The overall financial benefit of this practice has made ADDS a contributor to the over \$4 billion dollar revenue of NCR.

The success of the Mentor product line has since been progressive. Through 1985, more than 4,000 systems were sold. In that year, while continuing production of the Mentor product line, ADDS manufactured two PCs for the desktop-computing marketplace. Shortly afterward, to meet the needs of small businesses and to spark an interest in larger Mentor systems, a PC version of ADDS' Mentor OS was made available. This software could run on the ADDS PC as well as the IBM PC and IBM XT, and handle up to three users simultaneously.

More recently, with the advent of smaller but increasingly powerful systems, ADDS extended their penetration in the low-price multiuser business systems marketplace with the introduction of a Mentor system based on the Intel 80286 microprocessor. These systems provided the much needed link between the smaller and less powerful PC and ADDS' line of high-performance systems. Called the Mentor 1720 and Mentor 1740, these 16-bit microcomputers support up to eight users without a significant reduction in processing speed.

Like most other ADDS systems, the Mentor 1700 series computers operate under the Mentor OS.

FUTURE PRODUCT LINES AND THE DIRECTION OF THE COMPANY

ADDS has been able to maintain competitive prices on its systems and plans to continue doing so by staying abreast of current technology. In the past, computer memory was an expensive part of the computer. To keep prices down, hardware and software were designed around a small amount of memory. Now, advancements in technology have subsequently led to lower costs, thus enabling ADDS to offer more efficient systems at comparatively low prices. In its newest product offerings, the company will be supplying up to 16Mb of RAM in each system.

Perhaps the main reason ADDS has been successful is because of its handling of microprocessor technology. ADDS maintained its ability to get the same performance from its 16-bit processors as its competition got from dedicated processors in the minicomputer range. Developing the maximum abilities of the processor enabled the company to offer systems with the same performance at prices almost 50% less than the competition's. As might be expected, ADDS plans to continue its efficient development methods in the present computer marketplace.

THOMAS J. DEAS

ARCHITECTURE OF MICROPROCESSORS*

This article presents a tutorial overview of the past, present, and future of microprocessors and describes the key elements of their structure and operation. It is intended to serve as a technical introduction to the rapidly expanding field of microprocessor and microcomputer technology and to provide an overview of what these elements are, what they can do, and how they do it. The origin and evolution, as well as the basic principles of operation, are discussed. Several different types of microprocessor are considered, and examples of their application in the solution of real-world problems are given.

INTRODUCTION

The microprocessor, an invention of the early 1970s, essentially incorporates the computational power of a computer in a package that can be held in the palm of one's hand. Microprocessors are currently being used extensively in lieu of conventional logic to reduce product cost, add more functions, and increase reliability through fewer part numbers, reduced hardware, and less complex packaging.

The microprocessor is viewed differently by different people. To an end user it might be just a black box, to a programmer it might be a smaller version of the central workings of a computer, or to a control logic designer it might be a logic device that continually rewires itself according to the instructions sent to it. No matter what the point of view or intended end use, the microprocessor must contain certain elements in order to perform the required tasks. What is commonly referred to loosely as a "microprocessor" usually consists of a small group of devices that combine to make up the control nucleus of a microprocessor-based system (see Fig. 1). It is this nucleus of necessary devices that we are primarily concerned with in this article.

Basic operation of the various microprocessor system elements is covered first, followed by a step-by-step description of the execution of instructions in a simplified program. A number of more intricate concepts are discussed under Advanced Concepts, and The Future takes a look at where microprocessors are going from here. Example 1 is a simple cash register control, whereas Example 2 details the more sophisticated master/slave microprocessor control of a robot.

In order to shed some light on the original purpose of the device and to show how it has since grown to fill other needs, details of the origin and evolution of the microprocessor are given in Appendix 1. The physical

*© Copyright 1985 by International Business Machines Corporation; reprinted with permission.

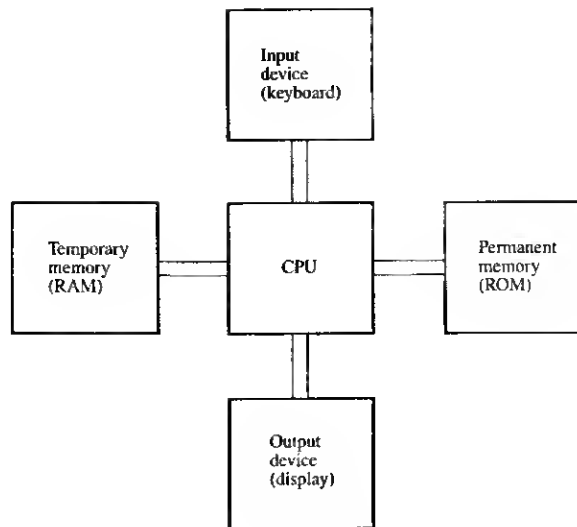


FIGURE 1 Simplified microprocessor block diagram.

packaging aspects of microprocessors and the details of operation of the various types of microprocessor memory devices are mentioned only briefly in the body of the article but are covered in some depth in Appendices 2 and 3. A bibliography has been included with references to more detailed tutorial and technical literature that covers the many aspects of microprocessors and how they are created and applied.

BASIC MICROPROCESSOR OPERATION

The primary control device in a microprocessor system is the CPU (central processing unit). Here, most of the decision making is performed through the execution of program instructions. These instructions are stored in memory, and the CPU fetches them as needed to perform the required task. Most instructions are very basic in nature; they direct simple operations, such as reading in data from an outside source, moving data from one storage register in the CPU to another, performing some logic or arithmetic function on the contents of a register, or writing data out to another part of the system. There are also instructions that perform more complex functions, but regardless of the complexity of a single instruction, it cannot accomplish even the simplest task entirely by itself. Individual instructions must be combined in some logical fashion to create a program to step the CPU through a series of basic operations that will, when taken as a whole, perform some useful function. Sometimes the program instructions are stored in the CPU itself, but more often they are stored in a separate memory device. A microprocessor needs memory for storing both instructions and data. The various types of solid-state memory modules used in microprocessor-based systems are described in Appendix 2.

A typical CPU chip consists of several separate logical sections, as shown in Figure 2. There must be a control ROM (read-only memory), a PLA (programmed logic array), or a random logic decoder (or some combina-

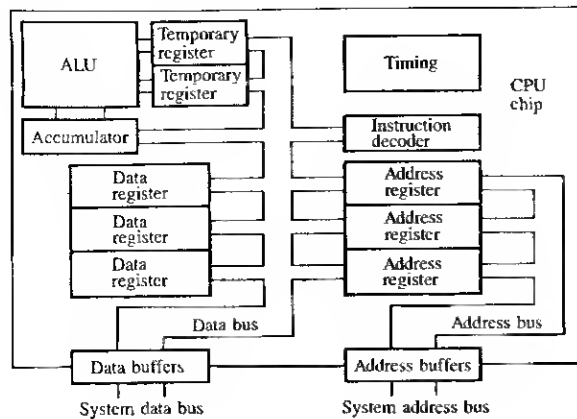


FIGURE 2 CPU block diagram.

tion of these devices) to decode the instructions one at a time and to direct the operation of the rest of the CPU chip. There is timing and sequence logic that steps each operation through in its proper order. There is an ALU (arithmetic logic unit) that performs basic arithmetic and logical operations on operands that are fed through it. There are normally a number of registers of various sizes located on the CPU chip itself. There are address pointer registers whose width depends on the size of memory the system is designed to handle and on whether the memory being addressed is in the CPU or external to it. There are data registers for storing and transferring data, and at least one of these registers is normally a special-purpose working register called an accumulator. The accumulator is involved in most of the data-oriented activity on the CPU. (The results of most of the ALU operations are sent to the accumulator, and its contents are quite often used as one of the operands.) Connecting all of these elements is a data path whose width is determined by the microprocessor word size. This data path, with bidirectional buffers at the boundary of the CPU chip, becomes the local system data bus and acts as the information path connecting all data-related elements in the system.

The contents of the active address pointer register generally follow a separate path to the boundary of the CPU chip, where it passes through buffers to become the local system address bus. A 16-bit address bus allows addressing of 65,536 (often referred to as 64K) separate memory locations, and a 20-bit address bus allows for over a million. In an effort to reduce the number of pins on the CPU package, some microprocessors multiplex some portion of the address bus and data bus on the same group of pins as they leave the CPU chip. This saves pins on the CPU but requires that extra hardware be added to create individual address and data buses to serve the rest of the system. This is of little consequence in larger systems, however, because the local address and data buses must be buffered again before being distributed to a large number of memory and peripheral chips, and the demultiplexing and buffering can both be done by the same devices. (The address bus is unidirectional only, out of the CPU, but the data bus is bidirectional and must be buffered in both directions.)

Microprocessors are sized or classified according to the number of binary digits or bits that they are able to handle at one time. An 8-bit

microprocessor generally has an 8-bit-wide data interface to the outside world and 8-bit internal data paths and registers. When there are exceptions to this, it is generally that the internal data path is wider in order to increase the functional capability of the CPU, or the interface to the rest of the system is narrower to reduce cost or the number of package pins. (There is at least one microprocessor with a 4-bit word size and a 1-bit interface off the CPU.) In 8-bit microprocessors, data are moved around in groups of 8 bits, which are referred to as "bytes." A byte is always 8 bits, and a "nibble" is half a byte or 4 bits, but a word may be of any size as defined by a particular microprocessor. The most common word sizes for microprocessors today are 4, 8, 16, and 32 bits.

A microprocessor needs a means of interfacing with other major elements in a control package or directly with the outside world. This is provided by interface elements or mechanisms called I/O (input/output) ports. These I/O ports may be located directly on the CPU itself or on separate chips and can handle input or output data in either serial or parallel format. Usually separate I/O ports are required for I/O data and for serial and parallel format, but there are sophisticated devices available today that may be programmed to handle many types of I/O, and even to handle automatically much of the protocol involved in some of the more involved serial communication formats. Parallel I/O ports may be designed for data input or data output only, or may handle bidirectional data. Some allow for the selection of input or output function on a bit-by-bit basis, but most ports are configured in groups of 4 or 8 bits. If the I/O port is located on a peripheral chip and not on the CPU, it is connected to the CPU via address, data, and control lines in a manner similar to that used for memory modules.

"Address modes" is a term that describes the different methods a CPU uses to address data stored for future use (normally in memory). Some CPUs have a number of different registers that may be used as address registers, and some of the address modes use these registers in different combinations. There are some address modes that do not use address registers as such. In "immediate" addressing, the data are included as part of the instruction and are therefore pointed to by the program counter. In "implied" addressing, the location of the data to be used is implied by the instruction itself. "Direct" addressing includes the address of the data as part of the instruction; "direct short" uses 8 bits of address (near the bottom or lower addresses of memory), and "direct long" uses 16 bits of address to access data anywhere in a 64Kb (kilobyte) range. In "register" addressing, the data are found in a register in the CPU, and in "register indirect" addressing, the instruction refers to a register that contains the address of the data in memory. "Indexed" addressing makes use of index registers in the CPU, and there is normally a method of automatically adding an offset to the contents of the index register before it goes out on the address bus (as for indexing into a data table). The index registers may also be caused to increment or decrement automatically by a specified amount every time their contents are used on the address bus.

Accessing program instructions is not usually considered one of the address modes in a microprocessor, because instructions are normally pointed to by a special-purpose address register on the CPU, called a "program counter." The program counter points only at instructions and is automatically incremented to the next instruction every time it is used. The program counter may also be force-loaded with an address, as in the case

of program jumps where the next instruction to be executed is not the next one stored sequentially in memory. The program counter might also be referred to as an instruction pointer register.

Another register not included in the address modes is the stack pointer. This is an address register that is normally loaded in the IPL (initial program load) routine to point to an area in RAM (random access memory) that has been set aside to store return addresses and miscellaneous data. This RAM area is referred to as a "stack," and its size determines the number of subroutines and interrupts that may be nested or overlapped. On a "call to subroutine" instruction, the CPU saves or pushes the address of the next instruction onto the stack by using the stack pointer as an address pointer and the contents of the program counter as the data to be written in memory. Next, the program counter is loaded with the address of the first instruction in the subroutine being called, and this causes a jump to that subroutine. (Subroutines are used when there is a series of instructions that are to be used over and over again in a program, as for a routine to read data in from a keyboard every time a key is pressed.) The last instruction in a subroutine is "return from subroutine," and it merely pops the return address off the top of the stack and places it in the program counter. Operation is then resumed immediately following the point in the program where the subroutine was last called. The stack pointer register is automatically incremented and decremented by the CPU so that it always points to the last entry or next available space. "Push" and "pop" instructions are available that allow the programmer to store temporary data on the stack or to save the contents of registers when a subroutine is called or an interrupt occurs. (The push instruction writes to the stack and pop reads from it.) The stack acts as a LIFO (last-in first-out) register, so any data must be popped or retrieved in the reverse order from that in which they were saved or pushed.

Some microprocessors, referred to as register based, have a large number of registers on the CPU, and the instructions specialize in operating efficiently on data in these registers. Another type, referred to as memory based, has fewer registers on the CPU, and the instructions are oriented more toward operating on data in memory. Still, a third variety has very few, if any, data registers on the CPU. Instead, there are pointers to memory where the "register" space actually resides. This fact is almost transparent to the programmer, except that there appear to be a number of complete sets of working registers that are easily accessible.

Tying together all the basic components in a microprocessor system generally involves a number of SSI and MSI logic chips. Buffers are required on the data bus to buffer the CPU from the electrically harsh environment on the system data bus and to increase the drive capability of the bus. Address decoders decode the higher-order address lines and provide chip select signals to activate the proper memory or I/O chips in response to a certain range of addresses. Multiplexing, or time-sharing, of address information and data on the same CPU pins requires that the system provide address latches to hold the address bus active throughout the entire memory read or write cycle. The basic system timing in a CPU is often derived from a crystal, and this usually requires a separate clock oscillator chip to convert the crystal frequency into clock signals suitable for the CPU chip. Some CPUs generate their own bus control signals, but more complex systems often utilize bus controllers to handle the read and write timing and other bus-related functions. What is really a fairly simple

microprocessor system can easily result in a printed circuit card with 20 or more devices on it. What is often referred to as a "computer on a chip" today is frequently in reality more of a "CPU on a chip." There is a constant effort, however, to reduce the number of chips required to form a working system. (Refer to "microcontrollers," discussed under Advanced Concepts.)

INSTRUCTION EXECUTION

In order to see just what a CPU does in carrying out its duties, we explore a short scenario of reading in some instructions and following the CPU through its paces. We assume that there is a program already in program memory to read the data at an I/O port, mask off some of the bits, and store the results in data memory. We also assume that this program starts at memory address 0, and that after it has been reset, this is where the CPU will look for its first instruction. For our example we assume that our microprocessor is accumulator based with isolated I/O and uses 8 data bits and 16 address bits. Figure 3 is a memory map for this program.

The first instruction in our program is a 2-byte INPUT instruction. The first byte is the op code, or operation code, that calls for the reading of the data from an I/O port. The second byte of the INPUT instruction is the number of the port that is to be read. In our example it is port 0. The second instruction is the AND instruction, which performs the masking of the bits. This is also a 2-byte instruction, with the first byte being the op code and the second byte being the immediate data to be used in the logical "and" operation. In our example we wish to mask off, or disregard, the high-order 5 bits of the input data, so the immediate data byte in the instruction is 07 (binary pattern 00000111). The third instruction is a MOVE instruction that takes our masked data and moves them out to memory. This instruction is 3 bytes long, with the first byte, as always, being the op code. The second and third bytes contain the 16-bit address

0007	HALT
0006	34
0005	12
0004	MOVE
0003	07
0002	AND
0001	00
0000	INPUT

FIGURE 3 Sample program memory contents.

of the memory location in which we wish to store the data. For our example we use memory address 1234. To prevent the CPU from continuing and doing anything after it has finished our example, we end the program with a HALT instruction that stops the CPU from executing any more instructions.

After the CPU has been reset by an external reset, the program counter contains all zeros, and its contents travel through address latches and out on the address bus to access memory location 0000. (Memory addresses are to be specified in our example by using four hexadecimal digits.) The CPU also sends out a memory read command so that the contents of the selected memory location are placed on the data bus by the memory module. The CPU reads the data bus, which now has the op code for the first instruction. Because this is the first operation after a reset, the CPU knows it must be reading in an op code so it places the first byte in the instruction decoder to be decoded. Upon decoding the instruction, the CPU discovers that it is a 2-byte instruction, and it must fetch the second byte. The program counter was automatically incremented after its contents were saved in the address latches, so it now contains address 0001. This address goes through the address latches and out to access memory location 0001 for the second byte. The second byte of the first instruction, which is 00 to represent port 0, goes back to the CPU by way of the data bus. Because this is an INPUT instruction, the CPU places the second byte of the instruction in the address latches as the lower half of the address of the I/O port. (The upper 8 bits of address are zeroes for all I/O operations.) Address 0000 goes out on the address bus, but this time the CPU sends out an I/O read command because it is reading an input port and not a memory location. The combination of address 0000 and the I/O read line being active causes input port 0 to place its contents on the data bus. (We might assume that the port is wired to some switches that we wish to monitor in a control application.) The CPU reads the data bus and places the data in the accumulator. (The accumulator is the destination for all INPUT instruction data and the source for all OUTPUT instruction data.) This concludes the execution of the first instruction, so the CPU prepares to read in the next op code.

The contents of the program counter again go out on the address bus, this time with address 0002 and a memory read signal. This causes the op code for the second instruction to go to the instruction decoder for decoding. The CPU discovers that this is a 2-byte AND instruction, and it sends the program counter contents of 0003 out on the address bus to fetch the second byte. The data in the second byte of the AND instruction are to be logically "anded" with the contents of the accumulator, so they are placed in a temporary holding register. The contents of the accumulator are copied into a temporary accumulator, and then both temporary registers are sent through the ALU, where they are "anded" together. The results of the ALU operation are returned to the accumulator, which now contains the low-order 3 bits of the data that were read in from port 0. "Anding" a number with 0 gives a result of 0. "Anding" a number with 1 leaves the number itself. Thus, our "anding" the accumulator with immediate data of 07 (00000111 in binary) has had the effect of masking the high-order 5 bits to zeros and leaving the low-order 3 bits unchanged. This concludes the execution of the second instruction, so the CPU prepares to read in the next op code.

The contents of the program counter go through the address latches and out on the address bus with address 0004 and a memory read signal.

This causes the op code for the third instruction to follow the data bus from memory into the CPU and then to the instruction decoder for decoding. The CPU discovers that it is a 3-byte MOVE instruction, and it sends the program counter contents of 0005 out on the address bus to fetch the second byte. The data in the second byte of the MOVE instruction are to be part of a memory address, so they are placed in half of a temporary address register in the CPU. The program counter was automatically incremented after its contents were saved, so it now contains address 0006. This address goes through the address latches and out to access memory location 0006 for the third byte of the instruction. The third byte of the MOVE instruction is the rest of the 16-bit memory address, so it goes to the other half of the temporary address register. The MOVE instruction is being used to move the contents of the accumulator out to data memory, so the temporary address register goes through the address latches to access memory location 1234. In this case, we are moving data out of the CPU, so the contents of the accumulator are placed on the data bus and the CPU issues a memory write command. This causes the data on the data bus to be latched into the memory location that is being addressed by the address bus and concludes the execution of the third instruction.

The HALT instruction is then read in from memory location 0007 and decoded. The CPU now remains in an idle state until it is given an external reset or an interrupt. No matter how long or involved a program may be, it is always executed one step at a time, as we have just seen (though, for ease of exposition, we did leave out a number of the details). There are microprocessors that can operate on more than one instruction at a time, and many of the instructions cause more activity than those we looked at, but it is always done one step at a time.

Programs to control microprocessors may be written in a number of different ways. The most basic of these is "machine language," which comprises the actual binary zeros and ones that the microprocessor responds to when it reads them in as instructions. The zeros and ones are fine for the microprocessor, but it is not comfortable for programmers to have to write programs using only numbers, so this is not done very often unless the application is expected to sell in extremely high volume.

The next level of programming is that of using "assembly language," where all of the binary machine instructions have been assigned names that represent their operation. Examples of these are AND, INPUT, ADD, JUMP, SUBTRACT, and so forth. The programmer writes a program using the individual instructions by name. (These names are referred to as mnemonics.) The mnemonics are then entered into a computer along with a program called an assembler, and the mnemonics are converted into the binary numbers that the microprocessor actually operates on. Assembly-language programs can be very efficient because the instructions are selected one at a time, but they also take a long time to write, and the programmer must be familiar with all the details of how each particular microprocessor handles each instruction. Assembly language is used today for most programs that are relatively short and are stored in ROMs or EPROMs (erasable programmable read-only memory) and sold in large volume.

Higher-level languages are available, which make it easier to write programs because the language sounds more like basic English statements than elemental operation steps in a computer. After a program has been written in a high-level language, it is entered into a computer along with a program called a compiler. There are different compilers for each of the high-level

languages (such as BASIC, Pascal, FORTRAN, etc.), and there are specific versions for each of the target microprocessors. The compiler lists or compiles the individual mnemonics required to execute each of the general program statements, and then its output is run through an assembler to create the actual machine code or instructions. High-level languages save time for the programmer but quite often make poor use of the microprocessor time and memory because the compiler does not always make optimum use of the instructions. Some programs are written in a high-level language, and then time-sensitive portions are rewritten in assembly language to optimize the operation of the overall system.

ADVANCED CONCEPTS

There are many different types of microprocessors available today. Some of them do not have all the features mentioned in the section on basic operation, but others have most or all of these features plus many more. Sometimes a number of powerful features are built into the CPU itself, or in some cases a rather ordinary CPU is applied in a system incorporating very sophisticated peripheral chips. The applications run the gamut from, for example, washing machine control or children's toys to word processing to real-time image processing and continuous-path machining.

One feature that might be incorporated in a CPU to increase its performance is "pipelining." An example of this may be seen in instruction decoding. In our simple example, we saw the instructions going directly from the data bus to the instruction decoder 1 byte at a time for decoding. Some microprocessors have instructions that are over half a dozen bytes long, and the CPU might have a queue for storing several bytes of instruction before they go to the instruction decoder. This saves CPU execution time in two ways. First, separate bus control logic on the CPU can be dedicated to keeping the instruction queue full so that the rest of the CPU may concentrate on executing the instructions and not on trying to make optimum use of the bus. (Simple microprocessors spend most of the time fetching instructions from memory and in the read/write data operations on the bus and not on actual arithmetic and logical operations that are performed within the CPU.) Second, the instructions may be predecoded to a certain extent in the queue so that some operations may overlap in time if they do not require the same portion of the CPU logic or depend on the results of a preceding instruction. Pipelining may be as simple as allowing the execution of one instruction to overlap the fetching of the next op code or as advanced as operating on several instructions at the same time. The objective, of course, is to execute instructions as rapidly as possible and to speed up the overall system operation. There are microprocessors today that do a high-speed fetch of a number of instructions and place them in an instruction cache or temporary storage location on the CPU chip itself. There they can be accessed more rapidly by the CPU than if they were left in the system memory. This speeds up the overall system operation considerably.

One of the trends today is toward more and more complex peripheral chips to handle such things as serial communication protocols, display screen control, parallel I/O with handshaking, DMA (direct memory access), floppy disk controllers, and multifunction timer/counters. These chips normally have a number of control registers that dictate how the chip functions in

the overall operation of the system. When electrical power is first turned on, these chips are typically unable to function without first receiving "operating instructions" from the CPU. The CPU, in its own IPL routine, encounters a series of instructions that tell it how to configure the peripheral devices in the system. It does this by sending a series of command words to the control registers in the peripheral devices themselves. These command words, which must normally be sent to the peripheral controller chip in a special sequence, are used to "customize" the operation of the chip to fit the application. After this initialization, the I/O chips automatically handle the data in the prescribed manner every time they are called upon by the CPU or by the peripheral device they are supporting.

The term direct memory access refers to an operation that transfers data directly to or from memory without the data being handled by the microprocessor chip itself. The DMA controller is an intelligent peripheral chip that must be initialized by the CPU and told where the data are to be moved from and where they are to be moved to. Once so initialized, the DMA controller takes control of the data bus, the address bus, and the control lines that handle memory read and write operations. The CPU is temporarily forced off the bus and must wait until the DMA operation is completed before it can regain control of the bus. DMA operations save time when large amounts of data are to be moved, because the DMA controller can automatically transfer data at the full bandwidth of the bus and does not need to keep reading in program instructions to tell it what to do next.

The microprocessors we have been talking about until now might be called "conventional" microprocessors. There is a type of microprocessor that is generally referred to as a single-chip microprocessor, or microcontroller. The main feature of these devices is that an attempt has been made to place the entire system on one chip. (This includes the CPU, RAM, ROM, clock oscillator, and I/O ports.) A microcontroller has enough on-chip ROM to hold a control program of reasonable size (i.e., 1K or 2Kb). Some microcontrollers are available in both masked ROM and UV-erasable EPROM versions (as described in the section on memory), and some have provisions for additional off-chip ROM or EPROM to allow for future expansion of an application program. Some microcontrollers also provide for off-chip data RAM, in addition to the on-chip registers and scratch-pad RAM. Some of the other features incorporated into microcontrollers to increase their control capability include Boolean processing, timers and counters (cascadable or with optional prescalers), A/D (analog-to-digital) conversion, PLLs (phase-locked loops), and the ability to handle serial I/O. Microcontroller applications today are expanding rapidly and will continue to do so in both high-end and low-end applications. Typical applications of the newer high-end versions are analog data processing or signal processing; many of the lower-end versions will be buried within electrical equipment where the presence of a CPU will probably not be recognized by the end user.

The coprocessor is another type of processor that has been developed to enhance the capabilities of a microprocessor CPU. An example of this is the NDP (numeric data processor). The NDP cannot operate in a stand-alone situation as the microprocessor CPU does but, typically, is connected to the same address and data bus as the CPU. The NDP monitors the instructions as they go to the CPU, and when an NDP instruction is encountered, the CPU releases the bus and allows the NDP to take over the bus and execute the instruction. The CPU may wait while the NDP performs

some floating-point operation, or it may continue on its own and then at a later time query the NDP for the results of the operation. The NDP instructions are intermixed with the CPU instructions, so the programmer need not be overly concerned that some instructions are being executed by the CPU and some by the NDP. Coprocessors speed up processing time because they are dedicated to performing specific tasks that would take much longer in the more versatile but slower CPU. They also save time by being able to operate in parallel with the main system CPU.

Interrupts are far from new (they have been available on microprocessors since the 8008), but they do require some attention to detail in both the hardware and the software design. The simple approach to interfacing with a peripheral device would be for the CPU to poll its controller. (Polling is software that periodically checks to see whether the device requires service.) The device might be a printer that is ready for more data to print or a tape reader that has data for the CPU. This method is all right for applications where time is not critical and the peripherals can wait for service. (The printer could theoretically wait all day for more data to print without causing a system problem.) The reader, on the other hand, could cause a problem. If the data were not read from it before more data were received, the new data would be written over the old data and would thus destroy the old data. In this case, the reader should be able to interrupt the CPU in whatever it is doing and request that the data be read in a timely manner.

Interrupts may come from just one device in a system or from a number of devices. In any case, the CPU must know where in memory to find the service routine for the particular device that has caused the interrupt. It is normally up to the requesting device or an interrupt controller chip to place this information on the bus. The CPU saves, on the stack, the address of the instruction it was about to read in when the interrupt occurred. It then picks up the interrupt information from the bus and calls the proper routine to service the device. The last instruction in the service routine is "return from interrupt," which gets the return address from the stack, and the CPU returns to what it was doing when the interrupt occurred. Some CPUs have duplicate sets of registers so that an entire new set of working registers may be switched to rapidly when an interrupt occurs. If there are to be many interrupt sources in a system, a priority interrupt controller chip is used. This device keeps track of the devices that have requested service and allows interrupts of a higher priority to interrupt the CPU while it is servicing devices of lower priority. Some of the newer microprocessors and microcontrollers have priority interrupt hardware and microcode built into the CPU itself in order to simplify system design and eliminate the need for a separate interrupt controller chip.

Bit-slice microprocessors are a specialized type of microprocessor used in very high-speed applications or where a specialized instruction set is required. The faster bit-slice microprocessors employ ECL (emitter-coupled logic), because this configuration of bipolar transistors can provide switching speeds in the subnanosecond range. The bit-slice microprocessor instruction set is determined by the system designer and is stored in a block of high-speed memory. (This technique is often referred to as microprogramming.) With microprogramming it is possible to optimize an instruction set for any required application or to emulate the operation of existing microcomputers. The ALU portion of the processor is available in "slices" that are typically 4 bits wide. A number of these slices may be combined to

build a processor that operates on larger word sizes (i.e., 12, 16, 20, or 32 bits). The advantages of a bit-slice microprocessor are the high speeds possible and the flexibility of being able to design your own customized instruction set. The disadvantage is the fact that the hardware and software must be defined in great detail by the system architect and cannot merely be picked off the shelf and used as can be done with the conventional microprocessor that has everything predefined by the manufacturer. The lower sales volumes of bit-slice microprocessors also tend to raise the overall cost of these systems.

Bus arbitration chips are available to help control access to the bus in multiprocessing operations, and memory management units allow a microprocessor access to more physical memory than its number of address lines would normally allow. The smaller-device geometries used in the newer microprocessors are allowing them to operate at higher and higher speeds, and features such as memory management are being incorporated into the CPU chip itself. Separate user and supervisor modes for system control, as well as support of trap instructions for error detection, have been available for several years, but the newer devices are providing much more of this type of support. The user/supervisor separation has recently been increased to four levels of protection. The concept of virtual memory, in which the program is unaware that some memory space is in main storage and some is in external storage, has been extended to the concept of a virtual machine. There is better support for coprocessors in the newer devices, and the register sets and instructions are becoming better suited to compiler-generated code. Most of the new processors utilize memory segmentation, but some still provide for a form of linear addressing. Many data types are supported, and some devices go as far as to provide a descriptor architecture in which the information stored is only one step from being self-identifying. The end result is that the newer microprocessors are taking on more and more of the attributes of the traditional main-frame computer.

The entire microprocessor field has been expanding at such a rapid rate that it is difficult, if not impossible, to keep up to date on all aspects of it. One area that has suffered has been that of terminology. New products have been introduced at a rate that has prevented the literature from ever completely catching up; as a result, much of the terminology in use today is imprecise and inconsistent. One cannot fall back on the conventional field of electricity and electronics as it existed and was taught in universities prior to the 1970s. Today a student's knowledge can become obsolete in the same year he graduates. The differences in documentation also hinder system designers when they must work with literature and data sheets from a number of different manufacturers and combine different technologies from different vendors into a working and reliable system. Another problem in the microprocessor field is that its early development was driven to an extent by hobbyists and toymakers, as well as by industrial designers and programmers, and some of the effects of their influence can still be seen today.

THE FUTURE

Microprocessors have already changed the way most of us live our daily lives at home, at work, and at play. Just over a dozen years ago micro-

processors did not exist, and today there are millions of them in operation in such unlikely places as a child's toy, the engine of a car, or an electric mixer in the kitchen. To state that they will continue to change our lives is like pointing out that the sun will come up tomorrow. What nobody knows today is where this will take us before the end of the century, because by then it is very likely that more microprocessors will be used in applications that do not even exist today than all of the microprocessors currently in use. The world is still trying to adjust to living with computers, and now we have a new breed that will fit under a thumbnail.

Some of the changes currently taking place are quite obvious. More peripheral functions are being added to the CPUs, and more intelligence is being added to the peripheral devices to the extent that very soon there will be cases where it will be difficult to distinguish one from the other. Higher-level language translators are being incorporated in silicon and will facilitate human communication with the new devices, and the devices themselves are being designed to operate more efficiently with the higher-level programming languages. Microcontrollers are being improved in function and in bit-handling ability; this will ease their introduction into an entirely new field of low-level applications, such as "smart" traffic lights, vending machines, and electrical appliances, which will become a natural part of our everyday lives. Microcontrollers are also being improved to handle higher-end applications, such as signal processing and specialized industrial controllers, and are taking on many of the same features that are being added to the newer CPU chips.

Many different names are applied in reference to the technology used today for the silicon chips themselves, but most are variations of NMOS (refer to Appendices 1 and 3). CMOS is now becoming faster in operating speed and smaller in size, and is increasing in popularity to the point where some manufacturers are introducing new parts only in CMOS. Another recent development is the radiation-hardening of the parts in order to increase their reliability in certain harsh environments being encountered in some applications.

Although microprocessor use will be quite visible in some areas, many of the new applications will not be as noticeable to the end user. (The motorist will be aware of the processor in the dashboard of his new car, but he may not know that there is actually a small CPU in the windshield wiper motor, fuel injector, voltage regulator, headlights, heater, and in each of the turn signal and brake lamps.)

The higher-end applications, such as data processing and industrial control (and even some of the personal computers), will be much more visible to the end user. A number of 32-bit devices are already available, and even larger word sizes will surely follow. This does not mean, however, that there will be a mass movement to the larger word sizes. The first microprocessors were 4-bit devices, and although they did not find their way into a large number of different applications, they still remain as the largest-selling microprocessors. Eight-bit microprocessors come in second as far as volumes shipped, but they do appear in the largest number of different applications today. Sixteen-bit microprocessors are now reaching their maturity and appear in an ever-increasing number of applications. The 32-bit devices are being well received, but faster and more versatile 8- and 16-bit devices will remain as the backbone of the microprocessor industry for some time to come (at least until there is a revolutionary breakthrough in the technology required to interconnect multiple devices).

To state simply where microprocessors are going from here, they are becoming more powerful and complex in function, faster in operating speed, and smaller in size; they will soon be used everywhere.

EXAMPLE 1: A CASH REGISTER CONTROL

Our first example is a simple system using a microcontroller to operate a cash register. The cash register was chosen as an example because until recently it was a purely mechanical device, and its function is well understood. The marketplace today requires that mechanical devices be simplified or eliminated in order for a product to remain competitively priced. The use of a touch-pad keypad and LED (light-emitting diode) display eliminates the mechanically operated keys and price display of the traditional cash register. A microcontroller was chosen because the application is a high-volume one and unit cost is of prime concern. Also, the application itself is well suited to the capabilities of a microcontroller. The cash register, which is used by a fast-food chain, has a special keypad for entering items by name, and a five-digit display for showing the total price of an order, the amount tendered, and the change due (see Fig. 4). As an added feature that would not have been possible with the old-style cash register, a running total of receipts is kept for transmitting to a host computer at the end of the day.

The microcontroller chosen for this application has 1Kb of on-board masked ROM for storing the application program and 64 bytes of on-board RAM for data storage. For prototyping, there is a second version available that has UV-erasable EPROM in place of the ROM and is compatible pin for pin with the masked ROM version. There are three I/O ports of 8 bits each

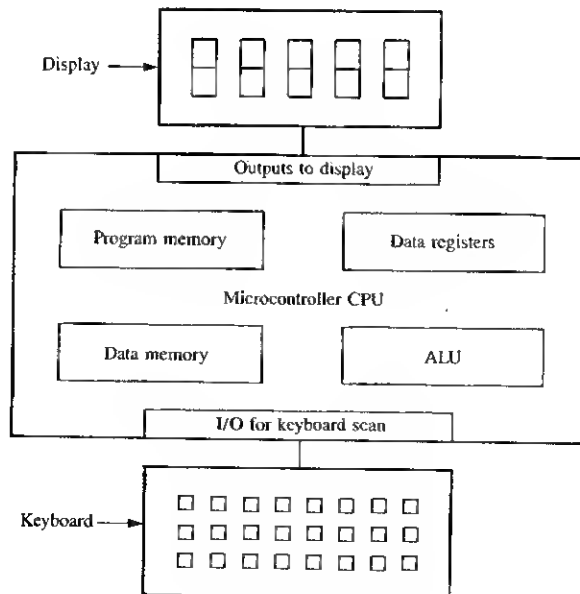


FIGURE 4 Cash register block diagram.

that provide 24 bits of I/O directly on the CPU chip. There is an accumulator, an 8-bit timer, eight 8-bit working registers, and an on-board stack that stores eight return addresses. No clock oscillator chip is required, as there are two pins on the CPU that accept a crystal directly as the timing element. There is also an interrupt pin that may be used to interrupt the CPU or that may be programmed to act as a polled input without actually interrupting the program in progress. Two "test" pins are available, one of which may be used either as an input or an output. A single 5-volt power supply is required.

The LEDs for the display are standard seven-segment displays (so named because of the seven individual segments that are used to form the number 8). The simple approach would be to use displays that accept BCD (binary-coded decimal) numbers from 0 to 9 and display the corresponding digits. This approach would require 16 of our 24 I/O lines (four display digits at 4 bits each). We have chosen instead to use LED displays that provide separate inputs for each of the seven segments plus one for the period (or decimal point). This means that we need eight data lines going to each display digit; but as well as displaying numbers, we will be able to form words such as "Err.," "CASH," and "DUE" by selecting proper combinations of the segments. The LEDs need not be energized 100% of the time to remain visible, so we multiplex them by connecting all similar inputs on a bus. (All segment ones are tied together, as are all segment twos, all periods, etc.) We illuminate the displays by strobing power to them one at a time, using four more of our 24 I/O lines. Going from four to five digits of display would add the possibility of words such as "HELLO" and "Error" to our vocabulary and would still only require a total of 13 I/O lines for the display. By multiplexing the display LEDs, we have saved three I/O lines and added a display digit, as well as reducing the total amount of power consumed by the display. We have also added a level of difficulty to the programming. Instead of writing to the display digits and leaving the data there until they are to change, we must now continually change the data and select lines in the proper sequence in order to keep the display lit. If we multiplex more than five display digits and also have to watch for new inputs from the keypad, the display will start to get dim.

The keypad for the cash register is of the EDS (elastic diaphragm switch) type and essentially has no moving parts to wear out. (It is also impervious to dirt and liquid spills.) There are 25 "keys," 22 of which are designated by the name of the products available. This reduces the possibility for operator error, as the inputs are by item name and the prices are automatically assigned by the microprocessor. If the prices change, a table in the program must be updated; if the items change, a new template may be placed over the keypad. The keys are arranged electrically in a 5×5 matrix and are strobed by 10 of the I/O lines. We have chosen to have the depression of any key cause an interrupt so that we scan the keypad only when necessary.

As the total price for each transaction is displayed, it is also added to a running total for the day that is being kept in RAM. If desired, a count can also be kept for any or all of the various items being sold. (A total of 255 can be accumulated in 1 byte, but 2 bytes allow the daily total to reach 65,535.) At the close of business (when the phone rates happen to be cheaper), the cash register may be connected to a long-distance line through a modem (modulator/demodulator). The daily records may then be transmitted to the home office, where they may be used for bookkeeping and/or inventory control purposes.

EXAMPLE 2: CONTROL OF A ROBOT

For our second example, we look at a more sophisticated system that uses a number of 16-bit microprocessors to control an industrial robot. The robot under consideration has eight axes or degrees of freedom. Each axis is driven by an electric motor and uses a microprocessor to control the speed of the motor. A ninth microprocessor acts as an overall supervisor to send data to the individual axis controllers, interface with the machine operator, and communicate with a host computer that oversees the operation of several robots. Each axis-control microprocessor is located on a separate printed-circuit card and is referred to as a slave. The master is located on a larger printed-circuit card, sends commands to the slaves, and also handles the miscellaneous robot control functions. Figure 5 shows the basic components in the system. We look first at a typical slave card and then at the master.

Each slave card has a 16-bit microprocessor with its associated logic and memory (refer to Fig. 6). Two EPROM sockets allow for up to 4Kb of program memory. Four $1K \times 4$ RAM modules provide 2Kb of scratch-pad RAM for storing program variables and a return address stack. There is an 8-bit DAC (digital-to-analog converter) to supply an analog (continuously variable) output used to control the speed of the axis drive motor. A 16-bit ADC (analog-to-digital converter) converts an analog input signal ranging from -10 volts to +10 volts into a 16-bit digital word representing the position of the robot axis in its range of travel. The microprocessor program on the slave card continually reads the axis position from the ADC. If the axis position does not match the last command position from the master, the microprocessor program causes the DAC to send a voltage

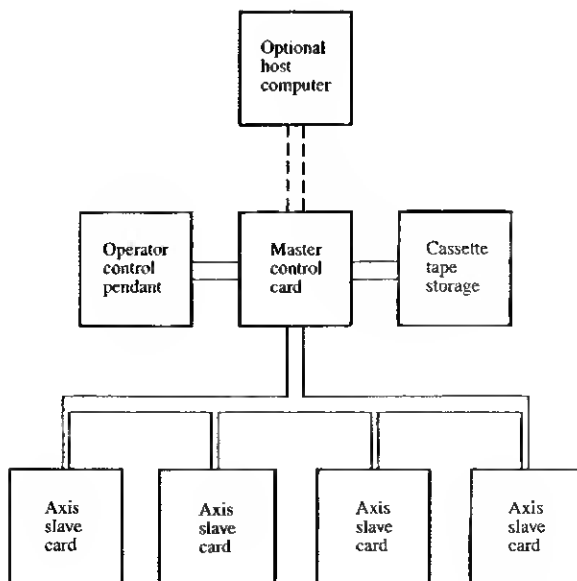


FIGURE 5 Robot simplified block diagram.

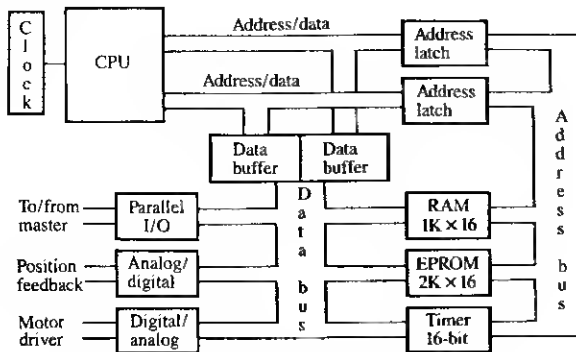


FIGURE 6 Slave card block diagram.

to the axis drive motor until the actual position matches the commanded position.

The slave card does not really have much to keep track of (the commanded position from the master, the actual position from the ADC, and the voltage to send out to the drive motor). It must, however, operate at a high rate of speed because when the robot axis is moving at high speed, the ADC input changes rapidly and the motor signal must be tuned to this for smooth operation of the robot. The portion of the program that handles the motor drive signal is quite concise and is written in assembly language for optimum use of execution time. The rest of the program handles communication with the master, system diagnostics, and so forth, and is not as critical for execution time or memory space. To save programming time, this portion of the slave program can be written in a higher-level language.

The master card is physically larger than a slave card and contains a 16-bit microprocessor with its own associated control logic and memory. Figure 7 is a simplified block diagram of the master card. The two EPROM sockets on the master card are larger than those on the slave card and allow for up to 16Kb of program memory. Eight $1K \times 4$ RAM modules provide 4Kb of scratch-pad RAM for storing data tables and a return address stack. Some additional control chips are used with the microprocessor on the master card to allow for a DMA operation with the RAM on the card and for the use of a numeric processor to perform high-level math functions. The application table that determines the various positions of each axis is stored in RAM on the master card and may be brought in slowly from a tape cassette or rapidly from another CPU by using DMA. The numeric data processor is used when the robot is not just moving from one point to another, as in a pick and place operation, but must maintain a specific path and speed, as in a welding operation. A high-level language can be used to write the program for the master card except for the routines that handle communications with the slaves. These routines determine the sequence of events on the communications bus and are written in assembly language.

The robot can be "taught" to perform an operation by leading it through the process steps with an operator control pendant. Each step is recorded in a table that is later followed by the machine as it performs the operation. This table is in RAM on the master card, so it may be changed at any time. The data are lost when power is turned off, so the information is

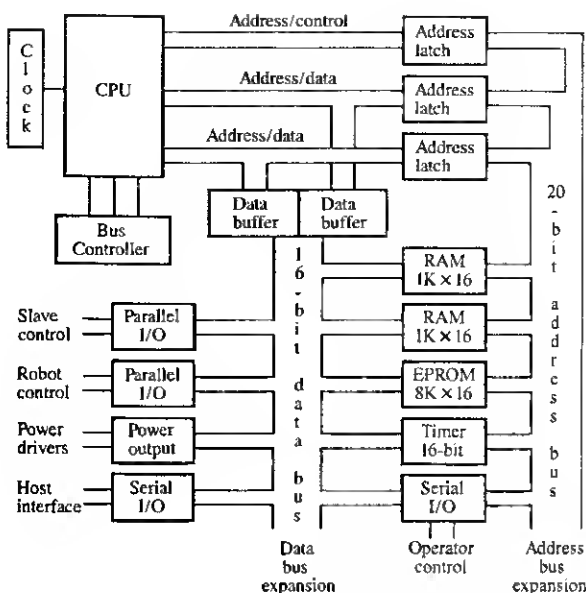


FIGURE 7 Master card block diagram.

stored by sending it out serially on a communication link to a tape drive or a host computer. This also allows the host computer to build up a table of robot operation steps and download it to the master card for execution.

A multimicroprocessor-based control for the robot was chosen as a cost-saving measure. The functions provided by the slave cards could have been implemented in conventional hardware, but the cost would have been greater because a large portion of the control function was eliminated from the hardware and performed with software. The functions provided by the master card could have been accomplished by a minicomputer. Here again there is a cost saving, because a minicomputer is stand alone and general purpose and the master card was designed specifically to control a multiaxis machine so that there is no unused hardware. Also, the master card is packaged on an 8 × 15-inch (20.32 × 38.10-cm) printed-circuit card inside the robot control unit, and the minicomputer would be in a moderately sized box of its own.

APPENDIX 1: EVOLUTION

The introduction of the transistor changed forever the physical aspects of computer circuitry. At first there was a constant effort to build smaller and smaller transistors until their size was reduced to the point where they became difficult to handle in manufacturing operations. Development effort was then directed toward placing as many transistors as possible on one piece of silicon, called a die or chip. This allowed circuit designers to create individual logic chips with ever-increasing levels of sophistication, and the resulting SSI (small-scale integration), MSI (medium-scale integration), and LSI (large-scale integration) random-logic devices were widely used by system designers in the proliferation of solid-state digital controls

that were introduced in the late 1960s and early 1970s (1). The point was then reached where LSI devices were becoming less general purpose and more dedicated to one specific application. At this point the introduction of new LSI devices began to slow down due to the diminishing return on investment required to develop and introduce new devices.

This was the environment in which the first microprocessor, or CPU on a chip, was born. The calculators being built at the time each used several logic chips containing up to 1,000 transistors apiece. Busicom, a Japanese manufacturer of calculators, approached Intel in mid-1969 with a request for 12 LSI chips for a new family of high-performance programmable printing calculators. At the time, Intel was producing MOS (metal oxide semiconductor) and bipolar RAMs and could place up to 2,000 transistors on a single chip. The Busicom design required 3,000 to 5,000 transistors on each chip and packages with 36 to 40 leads each.

The logic to control the printer, scan the keyboard, and update the display, as well as to perform the actual calculations, was to be incorporated on separate chips. The elemental instructions for performing the calculations were quite complex and were executed more in random logic than in ROM. In an effort to simplify the design, it was decided to reduce the complexity of the elemental instructions and thus make them more general purpose. With programs stored in ROM, these simple instructions could be used repetitively, not only to perform the required calculations but also to perform the logic functions required for the overall operation of the calculator.

A 4-bit-wide data path was chosen because it worked out well with the BCD coding being used in calculators at the time and allowed addressing of up to 16-digit positions for display or printing. The final version of the calculator control consisted of a four-chip set that eventually was called the MCS-4. It consisted of a 4-bit CPU chip, a ROM chip for program memory, a RAM chip for data memory, and a shift register chip for output expansion. The CPU chip ended up with about 2,300 transistors on a 16-pin substrate, becoming the 4004 that was advertised as a "microprogrammable computer on a chip" in November of 1971. The term microprocessor was first used for this device in 1972.

The CPU chip contained a 4-bit parallel adder, 16 index registers, an accumulator, and a pushdown stack for return addresses. There were 46 instructions, which included conditional branching, jump-to subroutine, indirect fetching, and support for both binary and decimal arithmetic. The 4004 had a 4-bit parallel bidirectional data bus that multiplexed address information and data. An eight-step instruction cycle was used to handle 8-bit instructions, and a 12-bit instruction address space allowed addressing of up to 4,096 bytes of program memory.

The microprocessor was therefore born out of necessity as a natural development in the progression of logic design and was initially intended more for the replacement of random logic than as an attempt to put a computer on a single chip.

Late in 1969, Computer Terminals Corporation (CTC) requested that Intel develop LSI chips for the registers of a new intelligent terminal. The 4004 instruction set had just been defined, and Intel proposed to CTC that the terminal control logic be implemented on one chip as a complete 8-bit processor. This design was not used in the CTC terminal, but it was implemented in silicon and became the first parallel 8-bit microprocessor. The CPU was introduced by Intel early in 1972 as the 8008. It had 45 instructions oriented toward the terminal requirement of character string

handling and also provided interrupt capability, a return address stack, and six general-purpose registers.

Late in 1972, Rockwell entered the microprocessor market with a 4-bit parallel processor called the PPS-4. The CPU had 50 instructions and was implemented in a PMOS (P-channel metal-oxide semiconductor) chip on a 42-pin substrate. Several other manufacturers introduced microprocessors in 1973 and 1974. Among these were Texas Instruments' TMS-1000 family of 4-bit PMOS processors, RCA's CMOS (complementary metal-oxide semiconductor) 1802, and National Semiconductor's IMP-16, which was a 16-bit five-chip bit-slice CPU.

The first three microprocessors marketed by Intel were implemented in PMOS. (There had also been a follow-on improvement to the 4004 called the 4040. It had 14 more instructions, a deeper stack, and more registers and memory addressing ability, as well as interrupt capability, all on a 24-pin substrate.) In 1974, Intel released an upgraded version of the 8008 and called it the 8080, the first of the second-generation microprocessors. The new processor consisted of 5,000 transistors implemented in faster NMOS (N-channel metal-oxide semiconductor) and had 30 more instructions than the 8008. The 8080 utilized a 40-pin package that made interfacing to it somewhat easier than with the 18-pin 8008, which required an additional 20 TTL (transistor-transistor logic) packages for memory and I/O interface. The 8080 had an 8-bit data bus and a 16-bit address bus, and the return-address stack was removed from the CPU and placed in RAM to allow virtually unlimited subroutine nesting. Decimal and double-precision arithmetic instructions missing from the 8008 were added to the instruction set, and the hardware provided the ability to handle vectored interrupts. (The interrupting device essentially tells the CPU where in program memory to find the proper interrupt service routine.)

Following the introduction of the 8080 in 1974, many new microprocessors have been introduced each year. (By 1976, the total was already up to 54.) The 6800 from Motorola in 1974 was the first to use a single +5-volt power supply, and Intel's 8048, in 1976, was the first 8-bit microprocessor to have a CPU, RAM, ROM, and I/O all on one chip.

Sixteen-bit microprocessors have been around in one form or another since 1974, but the first one to gain wide acceptance in the industry was the 8086 that Intel introduced in 1978. Texas Instruments' 9980, Intel's 8088, and Motorola's 6809 featured higher performance by utilizing 16-bit internal data processing but allowed simpler interfacing to the rest of the system by retaining an 8-bit external data bus. The trend today is to go to wider data paths (16-bit microprocessors are in widespread use, and 32-bit microprocessors are being introduced). There is also a strong effort to place more of the overall system function on the CPU in the form of operating systems in silicon, as well as auxiliary functions such as timers, interrupt controllers, communications controllers, and so forth, on the chip. Coprocessors are being used for high-level mathematical functions, such as floating-point operations, as well as intelligent I/O.

Some of the faster, special-purpose microprocessors have been implemented with bipolar transistors, but photolithographic and semiconductor processes have improved at a rapid rate, and most of today's microprocessors still utilize NMOS in one form or another. There is an increasing trend, however, to utilize CMOS in order to improve noise immunity and reduce power consumption.

The first microprocessors were used mainly for the replacement of random logic, as in the case of the printing calculator and the intelligent terminal previously discussed. They continue to be widely used in this type of application and have also branched into two other main areas. One is that of reducing the physical size of actual computers, as is evidenced by the rapidly growing personal computer market. The other is the introduction of intelligence into areas where it was not previously possible or practical. Some of the more popular examples of this may be seen in electronic toys, arcade games, automobiles, household appliances, robotics, instrumentation, and the ever-present video games.

In the 10 years since the introduction of the 8080, there have been improvements in speed and functionality, and instruction sets have been expanded to include more data-handling and program-control instructions, but the rudiments of the basic 8080 itself can still be seen in the majority of today's processors.

APPENDIX 2: MEMORY

A microprocessor system requires some type of memory for storing program instructions and also for storing constants and variable data. Instructions are often stored in ROM, and data are often stored in RAM. The information stored in a ROM is not volatile, that is, it is retained even when all electric power to the device is turned off. RAM, on the other hand, retains its data only while it is connected to a voltage source. ROM and RAM each come in several different types, which we now explore.

ROM may have its contents established during the manufacturing process, in which case it is called masked ROM. The user must specify the desired contents of the memory to the ROM manufacturer, who creates a photolithographic mask with the correct bit pattern. The masking of data into the ROM is an expensive and time-consuming process, but the finished parts themselves are relatively inexpensive. Masked ROM is therefore generally used in applications involving high-production items where many identical copies are required and where the program, or data being stored, is not likely to change.

Another type of ROM is the PROM (programmable read-only memory). The PROM is more expensive than masked ROM in large quantities, but it has an advantage in that it can be programmed by the manufacturer of the finished product in the plant and does not require that the information be essentially built into the parts by the semiconductor manufacturer, as is the case with the masked ROMs. The actual programming that loads the data or instructions into the PROMs is done on a device called a PROM programmer. PROM programmers are now available to meet a variety of needs. A simple and inexpensive one might be able to program a specific type of PROM or copy its contents into another one of the same type, where a more sophisticated and expensive model might be able to work with different types of PROMs and, perhaps, communicate with a terminal or a computer. Some microprocessor development systems have PROM programmers built in, so that after a program has been developed it can be loaded directly into a memory chip for debugging in the target system.

PROMs themselves come in several varieties. First is the "fusible-link" PROM that is personalized by selectively fusing metal links to change the

bit pattern stored in the device. The memory chip, as it is manufactured, contains all zeros or all ones, and the links are opened up one at a time to change the original data in one bit location at a time. This is a nonreversible process, so if changes must be made to the data, it generally means starting over again with a new PROM chip. Fusible-link PROMs do not typically have a large storage capacity and are used more in logic applications than for storing programs or data.

The second and, today, the most widely used type of PROM is the UV-erasable EPROM. As the name implies, a UV-erasable EPROM may be erased by a prolonged exposure to an ultraviolet light source. (There is a transparent window on the top of the package to allow the erasing light to reach the actual memory chip.) An EPROM may be programmed and erased many times before it finally fails to accept new data. The actual programming of the EPROM is done with a PROM programmer. In large quantities, the individual piece price of an EPROM is greater than that of a masked ROM, but it does have a number of advantages. Because an EPROM is erasable, it may be used in prototypes while the final code is being developed for eventual entry into a masked ROM. EPROMs may be gang-programmed in quantity and are therefore suited for applications that require repeat build but do not see enough replication to justify the cost of a masked ROM. Also, in the case of upgrading a product with new software or expanded capability, an EPROM can be exchanged and the old one sent back to be reprogrammed with the new code.

A third type of PROM is the EEPROM, which stands for electrically erasable programmable read-only memory. The EEPROM is similar to the UV-erasable EPROM, except that it may be programmed and erased electrically while in place in a system and need not be removed for the UV light and PROM programmer process. An EEPROM is also referred to as an EAROM (electrically alterable read-only memory).

RAM is used to hold information that is subject to change during the normal operation of a system. RAM is available in two basic types, known as "static" and "dynamic." In both cases, the information in the memory is lost when power is removed from the module and must be written back in each time power is applied. The difference between static and dynamic RAM is how the data are actually stored in the memory chip. A static RAM has flip-flop-type storage locations for each bit that is to be stored, and these flip-flops retain their information until it is written over or until power is removed from the device. In a dynamic RAM, the storage mechanism consists of a single transistor for each storage location. The information is stored as zeros or ones by charging or discharging a capacitor on the base of the storage transistor. Dynamic RAM is less expensive than static RAM, but the capacitors leak and must be recharged repeatedly. Circuitry known as "memory refresh logic" is required to keep recharging the proper capacitors. This is done automatically, but the logic takes up space on the circuit board and adds cycles to the process time of the system.

A newer type of RAM is the NOVRAM (nonvolatile random access memory). These devices contain an area of high-speed static RAM that is backed up on the same device by an identical array of EEPROM. The static RAM is used for normal read and write operations, and its contents may be transferred to the EEPROM very quickly if a power failure occurs. The data may then be retrieved from the EEPROM when the system is again

operational. These devices provide the benefits of both normal high-speed read and write operations and nonvolatile storage in the same package.

"Byte-wide memory" is a packaging scheme that is becoming popular in microprocessor-based systems. This scheme uses identical package pinouts for the same sizes of RAM, ROM, and EPROM. This allows EPROM to be used in the development stage of a project, and when the program has been finalized and debugged, ROMs may be ordered that directly replace the EPROMs with no wiring changes required to the system. Another advantage of having identical pinouts is that printed-circuit cards may be fabricated with standard prewired memory sockets that have been assigned addresses in the memory map. The end user may then populate the card with the type of memory needed for the particular system at hand, whether it is RAM, ROM, or EPROM. In byte-wide memory systems, package pinout is also considered when different sizes of memory are utilized. Smaller memory modules may be plugged into sockets intended for larger-capacity memory. In this case, care must be used to ensure that the proper pins on the module interface with the corresponding socket pins because the socket, which is able to support a larger memory, has more pins than the module.

Memory modules are available in many different sizes and are generally categorized according to the total number of data bits that may be stored. (A 4K RAM, for instance, has 4,096 individual bit positions.) The number of data lines on a memory chip is determined by the size of the words that may be stored, and the number of address lines is determined by the number of words. Ten address lines allow the addressing of 1,024 words, and eight data lines allow for an 8-bit word size. A memory manufacturer may refer to this as an 8K or 8,192-bit device, but in a microprocessor system it is more likely to be called a $1K \times 8$ memory (for the 1,024 words of 8 bits each). For larger word sizes, the address lines of several memory chips may be wired in parallel. For more words, the data lines and the lower-order address lines may each be wired in parallel, and the higher-order address lines are used to select or activate the proper memory chip. System memory may thus be tailored to almost any application by the proper selection and interconnection of memory modules.

APPENDIX 3: PACKAGING

The overall package size of a microprocessor is determined not by its computing power or the amount of on-chip logic it contains, but rather by the number of lines needed to interface the microprocessor itself with the rest of the system it is controlling or operating in. A microprocessor CPU consists of a small square of silicon, referred to as a die or chip, which contains the necessary solid-state logic circuitry. This chip is mounted on a substrate that acts as a chip carrier and contains the physical interface to the rest of the system. A typical microprocessor chip is about 0.2 mm thick and 2 or 3 mm square. The chips are fabricated from silicon wafers that measure up to several inches in diameter. The wafers go through a number of photolithographic and chemical process steps before finally being diced into individual chips. Each chip or die is then mounted on a substrate that is suitable for handling and for final assembly onto a card or board.

Individual microprocessor package sizes vary, but a typical one today has 40 pins, measures 1.5 cm wide by 5 cm long, and has two parallel rows

of 20 pins each. The double row of pins is evidenced in the name DIP (dual in-line package) that has been given to this particular type of package (also referred to as a DIL, for "dual in-line"). The individual pins are spaced 0.1 inch (0.254 cm) apart and the rows are spaced 0.6 inch (1.524 cm) apart. The inputs and outputs on the chip itself consist of metal pads that are spaced only mils apart around the periphery of the chip. Very fine wires are bonded to these pads to connect with metal traces on the substrate. These metal traces connect with the pins on the substrate, and the number of pins and traces needed is the determining factor in the size of the substrate. (In some of the newer, more sophisticated designs, especially those with 32-bit architectures, the size of the silicon chip is also becoming a determining factor.) The substrates may be made of ceramic or hard plastic and are hermetically sealed to prevent contaminants from contacting the silicon chip or interface wiring. The DIP packages are often referred to as modules and, to add to the confusion, quite often are loosely referred to as chips.

Some of the newer microprocessors are getting away from the DIP packaging in order to save printed-circuit board space and have gone to a square substrate that has rows of pins fastened to the bottom of the substrate. This type of package is called a pin grid array. There are also leadless chip carriers, which are square ceramic substrates that have no protruding pins. Connections are made to this device by contacting metal pads located around the periphery of the substrate in a process referred to as surface-mount technology, or SMT. (Leadless chip carriers are sometimes imprecisely referred to as flat packs.) Another method of fastening substrates to printed-circuit boards is via J-shaped leads that allow for different thermal coefficients of expansion for the substrate and the board.

SSI, MSI, and LSI are terms that relate to the amount of logic or number of "equivalent gates" that are present on one chip. A chip having fewer than 10 equivalent logic gates is referred to as SSI. MSI denotes chips with between 10 and 100 gates, and anything over 100 gates is known as LSI. VLSI is sometimes used for chips containing over 1000 gates in equivalent logic.

There are many names for the technology used in placing digital logic on semiconductor chips, but most are variations of bipolar and MOS. Bipolar transistors are typically fast and use more power than the field-effect transistors that make up the MOS devices. PMOS transistors use holes as majority carriers and are thus slower than the more popular NMOS transistors that use electrons as the majority carriers. CMOS chips utilize both NMOS and PMOS transistors on the same silicon substrate, with the result being a device that uses very little power and has a high immunity to electrical noise.

ACKNOWLEDGMENTS

The author wishes to thank Rex Dixon, Ray Floyd, Ed Galli, and Will Tracz for their support, encouragement, and suggestions in the preparation of this paper.

REFERENCE

1. Marcian E. Hoff, Jr., and Robert N. Noyce, "A History of Microprocessor Development at Intel," *IEEE Micro* 1, 8-11, 13-21 (February 1981). Note: This article contains a wealth of information concerning the environment into which the microprocessor was born; it includes history on all early publicized microprocessors, not just those developed by Intel. This article has been reprinted by Intel as "AR-173" (available from Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051).

BIBLIOGRAPHY

This bibliography has been selected as a small sampling from the author's personal library in an attempt to touch upon all areas of the broad field of microprocessors. There are selections specifically to cover history and some on technical details, but the majority were selected with the tutorial theme of this article in mind. A few references to basic computer architecture have been included as a bridge from computers to microcomputers to microprocessors, but programming references have been eliminated except for those directly applicable specifically to microprocessors. The bridge from digital discrete logic is covered quite thoroughly in the early literature on microprocessors. The listing is far from being complete or exhaustive, but it should contain material that will be helpful to readers of virtually any background. The arrangement is chronological, with the titles preceding the names of the authors. The earlier entries show how the industry got started in a particular area, and the more recent ones describe the latest developments. A number of books are included as overviews, along with a sampling of manufacturers' data books for exacting details. The main thrust, however, is on articles and papers, as they are generally short and to the point in a particular area, and one may scan the titles for subjects of particular interest.

1971-1973

- "Microprogramming Handbook," 2nd ed., Microdata Corporation, 644 E. Young St., Santa Ana, CA, November 1971.
- "MOS Integrated Circuits," National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, February 1972.
- "Microprogrammable Arithmetic Processor System—Orientation," National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, May 1972.
- "MCS-8 Microcomputer Set—8008 Users Manual," Rev. 4, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, November 1973.

1974

- "IMP-16C Application Manual," Publication No. 4200021C, National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, January 1974.
- "MCS-4 Microcomputer Set Users Manual," Rev. 5, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, March 1974.

- "MPS Microprocessor Series Users Handbook," *Manual No. DEC-08-UMPHA-A-PA*, Digital Equipment Corporation, Maynard, MA 01754, July 1974.
- "Microprocessor Design Series" (four reprints from *EDN* magazine, Vols. 18 and 19), R. H. Cushman (offered by *Design News* magazine, July 22, 1974).
- "Distributed Function Microprocessor Architecture," A. J. Weissberger, *Computer Design*, 13, 77-83 (November 1974).
- "PACE Users Manual," *Order No. IPC-16P/968X*, National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, December 1974.

1975

- "An Introduction to Microcomputers," A. Osborne, Adam Osborne and Associates, Inc., P. O. Box 2036, Berkeley, CA 94702, 1975.
- "ALTAIR 8800 Theory of Operation Manual and Schematics," Micro Instrumentation and Telemetry Systems, 2450 Alamo SE, Albuquerque, NM 87106, 1975.
- "F8 Circuit Data Book," Fairchild Semiconductor, Fairchild Camera and Instrument Corporation, 464 Ellis St., Mountain View, CA 94042, 1975.
- "How to Select and Use Microprocessors," *Pro-Log Microprocessor Users Guide*, Pro-Log Corporation, 2411 Garden Rd., Monterey, CA 93940, 1975.
- "LSI-11/PDP-11/03 Processor Handbook," Digital Equipment Corporation, Maynard, MA 01754, 1975.
- "Microprocessors," L. Altman, ed., McGraw-Hill Book Co., Inc., New York, 1971.
- "M6800 Microprocessor Applications Manual," Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, 1975.
- "M6800 Microprocessor Programming Manual," Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, 1975.
- "Series 1600 Microprocessor System," General Instrument Corporation Microelectronics, 600 W. John St., Hicksville, NY 11802, 1975.
- "Series 3000 Reference Manual," *Manual No. 98-221A*, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1975.
- "The Bugbook III—Microcomputer Interfacing," P. R. Rony, D. G. Larsen, and J. A. Titus, E & L Instruments Inc., 61 First St., Derby, CT 06418, 1975.
- "TMS 1000 Series MOS/LSI One-Chip Microcomputers," Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, 1975.
- " μ C Systems Directory" (foldout wall chart), R. M. Grossman, *EDN*, 20 (1975).
- "Understanding Microprocessors," D. Queyssac, published by Unwin Brothers Ltd., Old Woking, Surrey (undated); distributed by Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721.
- "990 Computer Family Systems Handbook," *Manual No. 945250-9701*, Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, 1975.
- "8080 Microcomputer Systems User's Manual," *Order No. 98-153C*, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, September 1975.

1976

- "An Introduction to Microcomputers—Volume I—Basic Concepts," A. Osborne, Adam Osborne and Associates, Inc., P. O. Box 2036, Berkeley, CA 94702, 1976.
- "An Introduction to Microcomputers—Volume II—Some Real Products," A. Osborne, Adam Osborne and Associates, Inc., P. O. Box 2036, Berkeley, CA 94702, 1976.
- "Dr. Dobbs' Journal of Computer Calisthenics and Orthodontia—Running Light Without Overbyte," Volume 1 (1976—entire year), J. C. Warren, Jr., ed., People's Computer Company, P. O. Box E, Menlo Park, CA 94025, 1976.
- "F8 User's Guide," *Manual No. 67095665*, Fairchild Micro Systems, Fairchild Camera and Instrument Corporation, 464 Ellis St., Mountain View, CA 94042, 1976.
- "Introduction to Microcomputers and Microprocessors," A. Barna and D. I. Porat, John Wiley & Sons, Inc., New York, 1976.
- "Microcomputer Design," D. P. Martin, Martin Research, 3336 Commercial Ave., Northbrook, IL 60062, 1976.
- "Microcomputers/Microprocessors Hardware, Software, and Applications," J. L. Hilburn and P. N. Julich, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.
- "Microprocessors and Minicomputers," B. Soucek, John Wiley & Sons, Inc., New York, 1976.
- "M10800 High Performance MECL LSI Processor Family," Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, 1976.
- "Scelbi '6800' Software Gourmet Guide and Cook Book," R. Findley, Scelbi Computer Consulting, Inc., 1322 Boston Post Road Rear, Milford, CT 06460, 1976.
- "Scelbi '8080' Software Gourmet Guide and Cook Book," R. Findley, Scelbi Computer Consulting, Inc., 1322 Boston Post Road Rear, Milford, CT 06460, 1976.
- "Software Design for Microprocessors," J. G. Wester and W. D. Simpson, Texas Instruments Learning Center, Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, 1976.
- "The ia7301 Computer in a Book," D. Guzman, IASIS, Inc., 815 W. Maude Ave., Sunnyvale, CA 94086, 1976.
- "8080 Programming for Logic Design," A. Osborne, Adam Osborne and Associates, Inc., P. O. Box 2036, Berkeley, CA 94702, 1976.
- "MCS6500 Microcomputer Family Programming Manual," *Publication No. 6500-50A*, MOS Technology, Inc., Valley Forge Corporate Center, Norristown, PA 19401, January 1976.
- "Microprocessor-Controlled Video Game Can Be Adjusted to Player's Skill," S. Davis, *Electronic Engineering Times*, 25-27 (April 12, 1976).
- "Microprocessor Basics" (25-part series), *Electronic Design*, 24-27 (April 26, 1976, through June 7, 1979).
- "16-Bit Processor Performs Like Minicomputer," A. Lofthus and D. Ogden, *Electronics*, 49 (May 27, 1976).
- "When Programming Microprocessors, Use Your Hardware Background," E. Lee, *Electronics*, 49 (July 8, 1976).

- "A Microcomputer Tailored for Multiprocessor Control Applications," G. Adams, T. Morgan, and J. Zarrella, *Control Engineering*, 23, 58-60 (September 1976).
- "'Universal' Development System Is Aim of Master-Slave Processors," R. D. Catterton and G. S. Casilli, *Electronics*, 49 (September 16, 1976).
- "EDN μ C Design Course," C. A. Ogdin, *EDN*, 21, 127-316 (November 20, 1976).
- "An Introduction to Microcomputer Software," M. G. Leonard, *Machine Design*, 48, 70-76 (November 25, 1976).
- "Putting a Microcomputer on a Single Chip," H. A. Raphael, *Computer Design*, 15, 59-65 (December 1976).
- "TMS 1000 Series Data Manual," Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, December 1976.
- 1977
- "An Introduction to Microcomputers—Volume 0—The Beginner's Book," A. Osborne, Adam Osborne and Associates, Inc., P. O. Box 2036, Berkeley, CA 94702, 1977.
- "Evaluation Kit Manual for the RCA CDP1802 COSMAC Microprocessors," Manual No. MPM-203A, RCA Solid State, 1998 Springdale Rd., Cherry Hill, NJ 07066, 1977.
- "How to Program Microcomputers," W. Barden, Jr., Howard W. Sams & Co., Inc., 4300 W. 62nd St., Indianapolis, IN 46268, 1977.
- Introductory Experiments in Digital Electronics and 8080A Microcomputer Programming and Interfacing," D. G. Larsen, P. R. Rony, and J. A. Titus, Howard W. Sams & Co., Inc., 4300 W. 62nd St., Indianapolis, IN 46268, 1977.
- "Microcomputer Handbook," C. J. Sippl, Petrocelli/Charter, New York, 1977.
- "Microcomputer-Based Design," J. B. Peatman, McGraw-Hill Book Co., Inc., New York, 1977.
- "Microprocessor Applications in Business, Science and Industry" (42 article reprints from a number of magazines), National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, 1977.
- "Microprocessor Architecture," M. Biewer, Pro-Log Corporation, 2411 Garden Rd., Monterey, CA 93940, undated.
- "Microprocessor Systems Handbook," Dr. D. P. Burton and Dr. A. L. Dexter, Analog Devices, Inc., P. O. Box 796, Norwood, MA 02062, 1977.
- "Microprocessors and Microcomputer Systems," D. H. Sawin III, Lexington Books, D. C. Heath and Co., Lexington, MA, 1977.
- "Microprocessors in Systems," R. H. Fish III, System Insights, P. O. Box 1, Austin, TX 78767, 1977.
- "Parallel Processing System (PPS) Microcomputers," Rockwell International Microelectronic Devices, P. O. Box 3669, Anaheim, CA 92803, 1977.
- "Practical Microcomputer Programming: The M6800," W. J. Weller, Northern Technology Books, P. O. Box 62, Evanston, IL 60204, 1977.
- "The 8080A Bugbook—Microcomputer Interfacing and Programming," P. R. Rony, D. G. Larsen, and J. A. Titus, Howard W. Sams & Co., Inc., 4300 W. 62nd St., Indianapolis, IN 46268, 1977.
- "TMS9940 Single Chip Microcomputer," J. D. Bryant and R. Longley (for presentation at Electro '77), Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, 1977.

- "Z80-Assembly Language Programming Manual," Zilog, Inc., 1315 Dell Ave., Campbell, CA 95008, 1977.
- "6800 Programming for Logic Design," A. Osborne, Adam Osborne and Associates, Inc., P. O. Box 2036, Berkeley, CA 94702, 1977.
- "A Logic State Analyzer for Microprocessor Systems," J. H. Smith, *Hewlett-Packard Journal*, pp. 2-11, 1820 Embarcadero Rd., Palo Alto, CA 94303 (January 1977).
- "Firmware for a Microprocessor Analyzer," T. A. Saponas, *Hewlett-Packard Journal*, pp. 12-15, 1820 Embarcadero Rd., Palo Alto, CA 94303 (January 1977).
- "Software for Microprocessors" (seven-part series), *Electronic Design*, 25 (January 4, 1977 through June 7, 1977).
- "Taking the Mystery Out of Micros," P. Roybal, *Machine Design*, 49, 80-83 (March 24, 1977).
- "Third-Generation Microcomputer Set Packs It All into 3 Chips," D. W. Sohn and A. Volk, *Electronics*, 50, 109-113 (May 12, 1977).
- "Analysis of Multiple-Microprocessor System Architectures," A. J. Weissberger, *Computer Design*, 16, 151-163 (June 1977).
- "16-Bit Microcomputer Is Seeking a Big Bite of Low-Cost Controller Tasks," J. D. Bryant, *Electronics*, 50 (June 23, 1977).
- "Unraveling the Mystery in User Microprogramming," R. Frankenberg, Part 1, *Mini-Micro Systems*, 10, 28-33 (June 1977); Part 2, *Mini-Micro Systems*, 10, 46-50 (July 1977); Part 3, *Mini-Micro Systems*, 10, 54-60 (September 1977).
- "EDN Software Design Course," C. A. Ogden, *EDN*, 22, 67-200 (June 5, 1977).
- "Microprocessor Networks," W. L. Spetz, *Computer*, pp. 64-70 (July 1977).
- "The First of the Third Generation Microcomputers," L. Goss, *Solid State Technology*, pp. 42-45 (July 1977).
- "Before Buying a Micro, Read the Menu," M. Biewer, *Machine Design*, 49, 68-74 (July 7, 1977).
- "A Microcomputer Course for Professionals," G. K. Holt, *Mini-Micro Systems*, 10, 36-40 (September 1977).
- "Acquiring Microcomputer Skills," C. A. Ogden, *Mini-Micro Systems*, 10, 42-48 (September 1977).
- "Comparisons and Trends in Microprocessor Architecture," E. E. Klingman, *Computer Design*, 16, 83-91 (September 1977).
- "Microelectronics" (reprint of eleven articles from *Scientific American*), W. H. Freeman and Co., 660 Market St., San Francisco, CA 94104, September 1977.
- "Making the Transition to Micros," C. A. Ogden, *Mini-Micro Systems*, 10, 32-37 (October 1977).
- "8X300 Reference Manual," Signetics Corporation, 811 E. Arques Ave., P. O. Box 409, Sunnyvale, CA 94086, October 1977.
- "EDN System Design Project," J. Conway, *EDN*, 22, 133-233 (November 20, 1977).
- "Microcomputer Overview" (eight chapters), C. A. Ogden, *Mini-Micro Systems*, 10, 32-127 (November-December 1977).
- "Is There a Tiger in Your Microcomputer?" *Electronic Business*, 3, 24-27 (December 1977).
- "Wrist Instrument Opens New Dimension in Personal Information," A. F. Marion, E. A. Heinsen, R. Chin, and B. E. Helms, *Hewlett-Packard Journal*, pp. 2-10, 1820 Embarcadero Rd., Palo Alto, CA 94303 (December 1977).

- "Microprocessor-Based Video Games," K. Li and A. Goldberger, *Electronic Design*, 25, 84-87 (December 6, 1977).
- "Stripping the Mystery from Microcomputers," L. Teschler, *Machine Design*, 49, 161-170 (December 8, 1977).

1978

- "A Guide to PL/M Programming for Microcomputer Applications," D. D. McCracken, Addison-Wesley Publishing Co., Reading, MA, 1978.
- "A Microprocessor Course," M. E. Fohl, Petrocelli Books, Inc., New York/Princeton, 1978.
- "Memory Design: Microcomputers to Mainframes," L. Altman, McGraw-Hill Book Co., Inc., New York, 1978.
- "Microcomputer 3870/F8 Data Book," Publication No. 79602, Mostek Corporation, 1215 W. Crosby Rd., Carrollton, TX 75006, 1978.
- "Practical Microcomputer Programming: The Z80," W. J. Weller, Northern Technology Books, P. O. Box 62, Evanston, IL 60204, 1978.
- "Series 8000 Microprocessor Family Handbook," National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, 1978.
- "The Bugbook VI—Introductory Experiments in Digital Electronics, 8080A Microcomputer Programming and 8080A Microcomputer Interfacing," D. G. Larsen, P. R. Rony, and J. A. Titus, E & L Instruments Inc., 61 First St., Derby, CT 06418, 1978.
- "The Complete Motorola Microcomputer Data Library," Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, 1978.
- "The Motorola MC3870 User's Manual," Manual No. M3870UM (AD), Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, 1978.
- "The Z-80 Microcomputer Handbook," W. Barden, Jr., Howard W. Sams & Co., Inc., 4300 W. 62nd St., Indianapolis, IN 46268, 1978.
- "Understanding Digital Computers," F. M. Mims III, Catalog No. 62-2027, Radio Shack (div. of Tandy Corporation), Fort Worth, TX 76102, 1978.
- "Z80 Instruction Handbook," N. Wadsworth, SCELBI Publications, Scelbi Computer Consulting Inc., P. O. Box 133, PP STN, Milford, CT 06460, 1978.
- "Z80 Programming for Logic Design," A. Osborne, J. Kane, R. Rector, and S. Jacobson, Adam Osborne and Associates, Inc., P. O. Box 2036, Berkeley, CA 94702, 1978.
- "8080/8085 Software Design with 190 Software Solutions," C. A. Titus, Howard W. Sams & Co., Inc., 4300 W. 62nd St., Indianapolis, IN 46268, 1978.
- "9900 Family Systems Design and Data Book," W. D. Simpson, G. Leucke, D. L. Cannon, Ph.D., and D. H. Clemens, Texas Instruments Learning Center, Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, 1978.
- "Handling Multilevel Subroutines and Interrupts in Microcomputers," J. F. Vittera, *Computer Design*, 17, 109-115 (January 1978).
- "Using Microprogramming to Standardize Minicomputer Interfacing," F. Grothman and J. Washburn, *Digital Design*, 8, 27-36 (February 1978).
- "Microcomputer Basics" (six-part series), *Electronic Design*, 26, 27 (February 1, 1978 through January 18, 1979).
- "Microprocessors Extend Scope of Automated Manufacturing," N. Sohrabji, *EDN*, 23, 101-106 (March 5, 1978).

- "Bit-Slice Microprogramming Saves Software Compatibility," S. Y. Lau, *EDN*, 23, 42-46 (March 5, 1978).
- "ICS Staff Goes Several Rounds with Microcomputers," J. Hall, J. Hickey, and R. Kuhfeld, *Instruments & Control Systems*, 51, 49-54 (April 1978).
- "ICS Staff Goes Several Rounds with Microcomputers—Part II," J. Hall, R. Merritt, B. Reynolds, and L. Zerone, *Instruments & Control Systems*, 51, 73-80 (May 1978).
- "Designing of a μ C Test Unit," W. W. Moyer, *Digital Design*, 8, 112-120 (May 1978).
- "Unified Buses Make the Peripheral IC/ μ C Connection," Dr. M. Smolin, D. Graves, K. Winter, and M. Schwartz, *Digital Design*, 8, 34-44 (May 1978).
- "The Intel 8086 Microprocessor: A 16-Bit Evolution of the 8080," S. P. Morse, W. B. Pohlman, and B. W. Ravenel, *Computer*, pp. 18-27 (June 1978).
- "Triple-Bus Architecture," J. Johnson, C. Kinnie, and M. Maerz, *Electronic Design*, 26 (July 19, 1978).
- "R6500 Microcomputer System Hardware Manual," Document No. 29650 N31, Rockwell International Microelectronic Devices, P. O. Box 3669, Anaheim, CA 92803, August 1978.
- "Complex Systems Are Simple to Design with the MC68000 16-Bit μ P," I. LeMair and R. Nobis, *Electronic Design*, 26, 100-107 (September 1, 1978).
- "Get Minicomputer Features at Ten Times the 8080 Speed with the 8086," G. Alexy and H. Kop, *Electronic Design*, 26, 60-66 (September 27, 1978).
- "Structured Programming Formulates μ P Program Logic," Dr. L. A. Leventhal, *Digital Design*, 8, 30-40 (October 1978).
- "EDN Software Systems Design Course," J. Hemenway, *EDN*, 23, 251-312 (November 20, 1978).
- "Cars and Computers Come Together," C. E. Wise, *Machine Design*, 50, 24-30 (November 23, 1978).
- "Second-Generation Microcontrollers Take On Dedicated-Function Tasks," J. Beaston and B. Schillhammer, *Electronics*, 51, 127-132 (November 23, 1978).
- "Integrating Peripherals into Processing Systems," R. J. Bufinger, *Computer Design*, 17, 77-83 (December 1978).
- "The Microprocessor Popularity Race," C. A. Ogden, *Mini-Micro Systems*, 11, 58-66 (December 1978).

1979

- "Basic Microprocessors and the 6800," R. Bishop, Hayden Book Co., Inc., Rochelle Park, NJ, 1979.
- "Microprocessor Software" (design-discipline reprint of 13 articles from *Machine Design*), Penton/PC, Penton Plaza, Cleveland, OH 44114, 1979.
- "Microprogramming Techniques with Sample Programs," S. J. Evans, Reston Publishing Co., Inc., Reston, VA, 1979.
- "Comparing Microprocessor Architectures," K. Rothmuller, *Mini-Micro Systems*, 12, 74-79 (January 1979).
- "Parallel Processor Architectures—Part 1: General Purpose Systems," K. J. Thurber, *Computer Design*, 18, 89-97 (January 1979).
- "Are Single-Chip Microcomputers the Universal Logic of the 1980s?" R. H. Cushman, *EDN*, 24, 83-89 (January 5, 1979).

- "As You Get to Know the 8086, Use Your 8-Bit Expertise," J. Hemenway and E. Teja, *EDN*, 24, 81-87 (January 20, 1979).
- "Parallel Processor Architectures—Part 2: Special Purpose Systems," K. J. Thurber, *Computer Design*, 18, 103-114 (February 1979).
- "Comparison of Selected Array Processor Architectures," S. P. Hufnagel, *Computer Design*, 18, 151-158 (March 1979).
- "New Options from Big Chips," J. Bayliss and J. McKevitt, *IEEE Spectrum*, 16, 28-34 (March 1979).
- "Troubleshooting Microprocessors with a Logic Analyzer System," R. Lorentzen, *Computer Design*, 18, 160-164 (March 1979).
- "Single-Chip N-MOS Microcomputer Processes Signals in Real Time," M. E. Hoff and M. Townsend, *Electronics*, 52, 105-110 (March 1, 1979).
- "CPU Brings 16-Bit Performance to 8-Bit Systems," J. Bartlett and R. Retter, *Electronic Design*, 27, 76-80 (March 15, 1979).
- "Coming: New Generations of Microcomputers," L. Teschler, *Machine Design*, 51, 108-114 (March 22, 1979).
- "A Microcomputer Industrial Control Interface Using I/O Modules," Manual No. MP716, Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, April 1979.
- "Designing a Microprocessor Driven Multipurpose Peripheral Controller," R. F. Binder, *Computer Design*, 18, 83-91 (April 1979).
- "Designing the LSI-11/23," G. Dulaney, *Mini-Micro Systems*, 12, 55-60 (April 1979).
- "Processor Architecture Anticipates Future Performance Requirements," R. E. Birney, *Computer Design*, 18, 71-79 (April 1979).
- "Software: Micros vs. Minis," K. Schroeder, *Digital Design*, 9, 20-26 (April 1979).
- " μ P Selection—Some Do's and Don'ts," P. Snigier, Part 1, *Digital Design*, 9, 28-32 (April 1979); Part 2, *Digital Design*, 9, 28-34 (May 1979).
- "How to Design Single Chip Microcomputers into Control Systems," W. Bottari, *Control Engineering*, 26, 69-72 (May 1979).
- "Micro Power Microelectronic Devices Data Catalog," Rockwell International Microelectronic Devices, P. O. Box 3669, Anaheim, CA 92803, May 1979.
- "Setting Up a Microprocessor Development Lab," B. Schweber, *Digital Design*, 9, 42-49 (May 1979).
- "Standard Modules Offer Flexible Microprocessor System Design," A. D. Hirschman, G. Ali, and R. Swan, *Computer Design*, 18, 181-189 (May 1979).
- "Comparing Architectures of Three 16-Bit Microprocessors," H. A. Davis, *Computer Design*, 18, 91-100 (July 1979).
- "How to Select the Optimum Microprocessor for Consumer and Industrial Controls," R. T. Barck, *Control Engineering*, 26, 106-110 (July 1979).
- "Microprocessor User's Guide," Pro-Log Corporation, 2411 Garden Rd., Monterey, CA 93940, July 1979.
- "A Preview of the Motorola 68000," A. I. Halsema, *BYTE*, 4, 170-174 (August 1979).
- "MC68000 Design Module User's Guide," Manual No. MEX68KDM (D2), Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, August 1979.
- "Wizards of Silicon Valley," G. Bylinsky and Z. Lane, *OMNI* (August 1979).

- "Zilog Z8000 Family Technical Overview," Zilog, Inc., 1315 Dell Ave., Campbell, CA 95008, August 1979.
- "16-Bit Microprocessors," S. Davis, *EDN*, 24, 70-85 (August 5, 1979).
- "MC6809 Course Notes," Motorola Technical Training, Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, September 1979.
- "Single-Chip 6801 Offers Versatility," J. J. Farrell III, Part 1, *Digital Design*, 9, 62-71 (September, 1979); Part 2, *Digital Design*, 9, 42-52 (October 1979).
- "70-Series Microprocessor Users Manual," *Manual No. μ PG-000001*, National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, September 1979.
- "Microprocessor Lab Teaches Operation and Troubleshooting," B. Bronson and M. Slater, *Hewlett-Packard Journal*, pp. 3-8, 1820 Embarcadero Rd., Palo Alto, CA 94303 (October 1979).
- "The 8086 Family User's Manual," *Manual No. 9800722-03*, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, October 1979.
- "EDN Advanced Software Systems Design Course," J. Hemenway and E. Teja, *EDN*, 24, 293-336 (October 20, 1979).
- "Learn to Apply the Power of the Z8002 by Studying a Small 16-Bit Computer," R. Korody and P. Alfke, *Electronic Design*, 27, 90-96 (October 25, 1979).
- "The Intel 8086," S. Ciarcia, *BYTE*, 4, 14-24 (November 1979).
- "Use a Systematic Procedure to Evaluate New μ Ps," J. Hemenway, *EDN*, 24, 185-193 (November 20, 1979).

1980

- "Crash Course in Microcomputers," L. E. Frenzel, Jr., Howard W. Sams & Co., Inc., 4300 W. 62nd St., Indianapolis, IN 46268, 1980.
- "MC68000 Cassette Training Tape," Motorola Technical Training, Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, undated.
- "MC68000 Course Notes," Motorola Technical Training, Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd, Austin, TX 78721, January 1980.
- "MC68000 16-Bit Microprocessor User's Manual," *Manual No. MC68000UM (AD2)*, Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, 1980.
- "Microelectronics Product Guide," General Instrument Corporation Microelectronics, 600 W. John St., Hicksville, NY 11802, 1980.
- "Microprocessor Basics" (design-discipline reprint of 11 articles from *Machine Design*), Penton/PC, Pentan Plaza, Cleveland, OH 44114, 1980.
- "The NS16000 Family of 16-Bit Microprocessors," National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, 1980.
- "The 8086 Book—Includes the 8088," R. Rectar and G. Alexy, OSBORNE/McGraw-Hill, 630 Bancroft Way, Berkeley, CA 94710, 1980.
- "The 8086 Primer—An Introduction to Its Architecture, System Design, and Programming," S. P. Morse, Hayden Book Co., Inc., Rochelle Park, NJ, 1980.

- "Primer on Microprocessor Development Systems," G. Nadler, *Electronic Products*, 23, 54-57 (January 1980).
- "Realtime Analyzer Aids Hardware/Software Integration," R. Francis and R. Teitzel, *Computer Design*, 19, 140-150 (January 1980).
- "Logic-State Analyzers Seek Out μ P-System Faults," G. Brock, *EDN*, 25, 137-140 (January 5, 1980).
- "The Promise of Analog μ Ps: Low-Cost Digital Signal Handling," R. H. Cushman, *EDN*, 25, 127-132 (January 5, 1980).
- "Forum on Testing Microprocessors," A. Mendelsohn, *Electronic Products*, 23, 35-40 (February 1980).
- "Microprocessor Troubleshooting Techniques," D. Wiseman, *Electronics Test*, 3, 42-48 (February 1980).
- "Communications in Distributed Systems--Part 1: Interfacing Techniques," M. G. Gable, *Computer Design*, 19, 30-34 (February 1980); "Part 2: Common Bus and Shared Resource Access Schemes," *Computer Design*, 19, 14-27 (March 1980); "Part 3: Communication Protocols and System Design Considerations," *Computer Design*, 19, 14-22 (April 1980).
- "To Get to Know Analog μ Ps, Simulate Simple Examples," R. H. Cushman, *EDN*, 25, 137-146 (February 5, 1980).
- "The MC68000--A 32-Bit μ P Masquerading as a 16-Bit Device," R. Grappel and J. Hemenway, *EDN*, 25, 127-134 (February 20, 1980).
- "Meeting EPROM Requirements of Advanced Microprocessors," T. Coffman, *Computer Design*, 19, 212-220 (March 1980).
- "Designer's Guide to: Testing and Troubleshooting μ P-Based Products," M. J. Weisberg, *EDN*, 25, 175-214 (four parts) (March 20, 1980).
- "Bit-Slice Design Approaches," H. Brineen, *Computer Design*, 19, 184-191 (April 1980).
- "Fiber Optics Successfully Links Microcomputers," S. Evans and J. Herman, *Digital Design*, 10, 36-37 (April 1980).
- "Memory-Management Units Help 16-Bit μ Ps to Handle Large Memory Systems," J. Hu, H. Yonezawa, and B. Pueto, *Electronic Design*, 28, 128-135 (April 26, 1980).
- "Microcontroller Doubles as Boolean Processor," B. Koehler, *Electronic Design*, 28, 57-62 (May 24, 1980).
- " μ P-Controlled 'House of the Future' Serves as a Product-Development Lab," W. Twaddell, *EDN*, 25, 65-74 (June 20, 1980).
- "Pick a Computer Language That Fits the Job," M. Schindler, *Electronic Design*, 28, 62-78 (July 19, 1980).
- "Indexed Mapping Extends Microprocessor Addressing Range," I. LeMair, *Computer Design*, 19, 111-118 (August 1980).
- "Word Processing System Design for High Throughput," P. D. Cherry, *Computer Design*, 19, 95-99 (August 1980).
- " μ P-Based Product Design Starts with μ P Selection," M. Mihalik and H. Johnson, *Electronic Design*, 28, 119-125 (September 1, 1980).
- "Compare the Newest 16-Bit μ Ps to Evaluate Their Potential," R. Grappel and J. Hemenway, *EDN*, 25, 197-201 (September 5, 1980).
- "Microcomputer Development Systems," A. Santoni, *EDN*, 25, 141-151 (September 5, 1980).
- "Development of Microprocessor Software," M. Rooney, *Design News*, 36, 81-89 (November 17, 1980).

1981

- "An Introduction to ASM86," *Order No. 121689-001*, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1981.
- "iAPX 432 General Data Processor Architecture Reference Manual," P. Tyner, *Manual No. 171860-001*, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1981.
- "Microprocessors Data Manual," *Manual No. DL-120*, Motorola Semiconductor Products Inc., 3501 Ed Bluestein Blvd., Austin, TX 78721, 1981.
- "NSC800 Microprocessor Family Handbook," National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, 1981.
- "STD Methods for Modularity," R. Born (presented at Electro 1981), Pro-Log Corporation, 2411 Garden Rd., Monterey, CA 93940, 1981.
- "TMS9900 16-Bit Microprocessor Family," *Manual No. CL-483A*, Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, 1981.
- "Requirements for High-Performance Microcomputers," J. Gorin and L. Stern, *Mini-Micro Systems*, 14, 127-136 (March 1981).
- "Multitasking Executive Speeds 16-Bit Micros," J. M. Irwin, *Electronic Design*, 29, 131-135 (March 5, 1981).
- "Optimizing Microprocessor Input/Output Techniques," L. E. Costlow, *Computer Design*, 20, 151-160 (April 1981).
- "A Tale of Four μ Ps: Benchmarks Quantify Performance," R. D. Grappel and J. E. Hemenway, *EDN*, 26, 179-265 (April 1, 1981).
- "Built-In Test Capabilities Could Cure μ P-Based System Ills," D. Jones, *EDN*, 26, 105-109 (April 15, 1981).
- "Distributed Intelligence vs. Centralized Logic," P. L. Alker, *Mini-Micro Systems*, 14, 103-115 (May 1981).
- "iAPX 186 Microprocessor Architecture Overview," J. Klovstad and S. Kopel, Intel Corporation, 3065 Bowers Ave., Sanat Clara, CA 95051, May 1981.
- "The Impact of 16-Bit Microprocessors on Software Development Tools," M. J. Elmore, D. Miller, and J. Schwabe, *Computer Design*, 20, 111-115 (June 1981).
- "Understand Emulator Use to Increase Prototyping Skills," M. Mihalik and B. Francis, *EDN*, 26, 121-128 (June 10, 1981).
- "Forum on 8 and 16-Bit Microprocessors," A. Mendelsohn, *Electronic Products*, 24, 47-53 (June 15, 1981).
- "Microprocessor Applications Reference Book," Zilog, Inc., 1315 Dell Ave., Campbell, CA 95008, July 1981.
- "Memory-Management Chip Masters Large Data Bases," D. L. Collins and C. M. Collins, *Electronic Design*, 29, 115-121 (August 20, 1981).
- "SDLC Interface Mates M6800 Peripheral to 8086," S. Yakobovitch, *Computer Design*, 20, 169-180 (September 1981).
- "Silicon Operating System Standardizes Software," C. McMinn, R. Markowitz, J. Wharton, and W. Grundmann, *Electronics*, 54, 135-139 (September 8, 1981).
- "16-Bit- μ P Benchmarks—An Update with Explanations," W. Patstone, *EDN*, 26, 169-203 (September 16, 1981).
- "Microprocessor Data Manual," D. Bursky, *Electronic Design*, 29, 77-175 (November 26, 1981).

1982

- "CMOS—LSI Microprocessors—Memories—Peripherals," RCA Solid State, 1998 Springdale Rd., Cherry Hill, NJ 07066, 1982.
- "COPS Microcontrollers Databook," National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, 1982.
- "Peripherals Technical Overview," Zilog, Inc., 1315 Dell Ave., Campbell, CA 95008, 1982.
- "Texas Instruments Semiconductor Products Master Selection Guide," Manual No. SCG682, Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, 1982.
- "8086/8088 16-Bit Microprocessor Primer," C. L. Morgan and M. Waite, BYTE/McGraw-Hill, 70 Main St., Peterborough, NH 03458, 1982.
- "As μ P/ μ C Chips Mature, Support Chips Proliferate," R. H. Cushman, EDN, 27, 155-202 (January 6, 1982).
- "Introduction to the iAPX 286," Order No. 210308-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, February 1982.
- "16-Bit- μ P Peripheral ICs Provide Datacomm Support," D. R. Snyder, EDN, 27, 181-190 (February 17, 1982).
- "Memory Protection Moves onto 16-Bit Microprocessor Chip," P. Heller, R. Childs, and J. Slager, *Electronics*, 55, 133-137 (February 24, 1982).
- "Microprocessor Peripherals UPI User's Manual," Order No. 210317-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, April 1982.
- "Software in Silicon: More Than a Program in ROM," A. Verhalen, *Electronic Design*, 30, SS33-SS36 (May 13, 1982).
- "16-Bit μ P Crams Peripheral Support on Chip," J. Klovstad, G. M. Gatlin, and T. Zingale, *Electronic Design*, 30, 191-196 (June 10, 1982).
- "Microprocessors" (special issue on microprocessor systems development), *IBM Journal of Research and Development*, 26 (July 1982).
- "Technology Profile," D. Bursky, *Electronic Design*, 30, 83-94 (October 14, 1982).
- "Integrating Memory Management into the CPU," G. Alexy, B. Childs, and J. Crawford, *Electronic Products*, 25, 55-62 (October 25, 1982).
- "Multi-User Systems from Advanced Processor Chips," R. M. Schell, *Computer Design*, 21, 149-158 (November 1982).
- "Multiprocessing Improves Robotic Accuracy and Control," P. Gupta, *Computer Design*, 21, 169-176 (November 1982).
- "Fundamentals of Microprocessors," E. Lee, *Telephone Engineer & Management* magazine (November 1, 1982).

1983

- "A Programmer's View of the Intel 432 System," E. I. Organick, McGraw-Hill Book Co., Inc., New York, 1983.
- "Bipolar Microprocessor Logic and Interface Data Book," Advanced Micro Devices, Inc., 901 Thompson Pl., P. O. Box 453, Sunnyvale, CA 94086, 1983.
- "iAPX 286 Hardware Reference Manual," Order No. 210760-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1983.
- "iAPX 286 Operating Systems Writer's Guide," Order No. 121960-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1983.

- "iAPX 286 Programmer's Reference Manual," Order No. 210498-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1983.
- "iAPX 86/88, 186/188 User's Manual," Order No. 210911-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1983.
- "Introduction to the 80186 Microprocessor," K. Shoemaker, Order No. 210973-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1983.
- "Microprocessor and Peripheral Handbook," Order No. 210844-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1983.
- "MOS Microprocessors and Peripherals Data Book," Advanced Micro Devices, Inc., 901 Thompson Pl., P. O. Box 453, Sunnyvale, CA 94086, 1983.
- "MOS Microprocessor Data Manual 1983," Signetics Corporation, 811 E. Arques Ave., P. O. Box 409, Sunnyvale, CA 94086, 1983.
- "NCR/32-000 32-Bit Microprogrammable Microprocessor," NCR Microelectronics Division, Colorado Springs, CO, 1983.
- "NS16000 Databook," National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, 1983.
- "TMS32010 User's Guide—16/32-Bit Digital Signal Processor," Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, 1983.
- "An Architectural Comparison of 32-Bit Microprocessors," A. Gupta and H. D. Toong, *IEEE Micro*, pp. 9-50 (February 1983).
- "Mostek 1983 Computer Products Data Book," Mostek Corporation, 1215 W. Crosby Rd., Carrollton, TX 75006, March 1983.
- "Designing with Microprocessors: The Modular Approach," M. Biewer, *Telephone Engineering & Management*, 37, 67-69 (March 1, 1983).
- "Computerizing the Car," W. Brown, *SKY* magazine, pp. 24-33 (October 1983).
- "Microprocessors: Speed Up, Price Down, and CMOS Everywhere," S. Bassett, *Computer Design*, 22, 177-187 (October 1983).
- "Advanced Features Squeeze onto Processor Chip," J. Slager, *Computer Design*, 22, 189-193 (October 1983).
- "The M68000 Educational Computer Board," R. W. Floyd, *BYTE*, 8, 324-336 (October 1983).
- "Examine Architectures When Evaluating μ Ps," K. Christian, *EDN*, 28, 193-202 (October 13, 1983).
- "EDN's Tenth Annual μ P/ μ C Chip Directory," R. H. Cushman, *EDN*, 28, 111-256 (November 10, 1983).
- "Educational Board Computer Teaches Engineers the Microprocessor Game," M. Gallagher, *Electronic Products*, 26, 115-121 (November 17, 1983).
- "Bipolar Arithmetic Chip Speeds 68000's Math Throughput," V. J. Coli, C. Hastings, S. Rajpal, and R. W. Blasco, *EDN*, 28, 179-193 (November 24, 1983).

1984

- "Distributed Control Modules Databook," Manual No. 230973-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1984.
- "Guide to Using the Distributed Control Modules," Manual No. 146312-001, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1984.
- "How to Use Surface Mount Technology," J. Mullen, Manual No. SSYZ001, Texas Instruments Inc., P. O. Box 225012, MS-54, Dallas, TX 75265, 1984.

- "MC68020 32-Bit Microprocessor User's Manual," *Motorola Manual No. MC68020UM (ADI)*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
- "Microcontroller Handbook," *Order No. 210918-002*, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1984.
- "Microsystem Components Handbook" (two-volume set), *Order No. 230843-001*, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1984.
- "Series 32000 Instruction Set Reference Manual," *Publication No. 420010099-001B*, National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, 1984.
- "Software Handbook," *Order No. 230786-001*, Intel Corporation, 3065 Bowers Ave., Santa Clara, CA 95051, 1984.
- "8088 Family on the STD Bus," *System Designer's Guide, Manual No. Ziatechniques 5*, Ziatech Corporation, 3433 Roberto Ct., San Luis Obispo, CA 93401 (undated).
- "Introduction to the NS16000 Architecture," National Semiconductor Corporation, 2900 Semiconductor Dr., Santa Clara, CA 95051, February 1984.
- "STD Bus Technical Manual and Product Catalog," Pro-Log Corporation, 2411 Garden Rd., Monterey, CA 93940, February 1984.
- "8X300 Family Capability Manual," Signetics Corporation, 811 E. Arques Ave., P. O. Box 409, Sunnyvale, CA 94086, March 1984.
- "Abundant Board-Level μ Cs Pack Increased Processing Power," D. Powers, *EDN*, 29, 213-220 (April 19, 1984).
- "Versatile Serial Protocol for a Microcomputer-Peripheral Interface," D. C. Stanley (presented at Mini/Micro Northeast, May 1984), RCA Solid State, Rte. 202, Somerville, NJ 08876.
- "VLSI-Based LAN-Controller Chip Eases μ P-to-Network Interface," R. H. Cushman, *EDN*, 29, 207-220 (May 3, 1984).
- "Synchronous 32-Bit Backplane Buses Open Up Distributed-System Design," B. Nicholson, *EDN*, 29, 75-86 (June 14, 1984).
- "Thirty-Two Bit Micros Power Workstations," N. Mokhoff, *Computer Design*, 23, 97-112 (June 15, 1984).
- "Serial Backplane Suits Multiprocessor Architectures," M. Webb, *Computer Design*, 23, 85-96 (July 1984).
- "32-Bit Extension to the 68000 Family Addresses 4G Bytes, Runs at 3 MIPS," W. Twaddell, *EDN*, 29, 75-77 (July 12, 1984).
- "32-Bit Processors Pack Mainframe Muscle," J. Javetski, *Electronic Products*, 27, 49-55 (July 16, 1984).
- "Semiconductor Memories: Density and Diversity," T. Williams, *Computer Design*, 23, 105-116 (August 1984).
- "Circuit Density and Speed Boost Tomorrow's Hardware," J. Bond, *Computer Design*, 23, 210-225 (September 1984).
- "Hardware/Software Problems Yield to Today's Logic Analyzers," K. Lowe and M. Van Hook, *EDN*, 29, 203-208 (September 6, 1984).
- "Memory-Management Varieties Suit Different Application Areas," D. Phillips, *EDN*, 29, 135-143 (September 6, 1984).

ROBERT C. STANLEY

ARTIFICIAL INTELLIGENCE

INTRODUCTION

Artificial intelligence (AI) is now a widely used term associated both with computer-related research and with many commercial applications of computers. Nevertheless, the term is not well-defined. It is usually associated with computers, rather than, say genetic engineering, and it is thus generally agreed that "machine intelligence" would be a better term. But AI, the term claimed to have been coined by John McCarthy at the first AI meeting in 1956 at Dartmouth College, remains the popular choice.

AI refers to the efforts to program computers so that they will behave intelligently or will be able to do tasks that are normally assumed to require intelligence when humans do them. There are many topics that are considered to be part of AI, for example, game playing, theorem proving, planning, searching, natural language understanding, machine learning, knowledge-base management, and expert systems.

Although AI has been almost exclusively a research topic, in recent years, certain aspects have been seen to be sufficiently well-understood that they are ready for commercial application; knowledge bases, limited natural language interfaces and, above all, expert systems constitute the bulk of commercial AI.

The dramatic shift of both financial support and interest to AI in the last few years has been driven primarily by the potential applications of expert systems, such as R1 [1], which can perform highly limited but complex tasks, in this case configuring computer systems, as well as or better than the corresponding human experts.

A second reason for the recent expansion of interest in AI is due to the fact that many conventional applications of computers (e.g., missile guidance) were seen to be severely limited unless they were given some intelligent capability. So the move to AI is also a result of reaching the limitations of non-AI computer applications.

As a result of this explosive growth, the field of AI is currently less well-defined or easily characterized than ever before. We may view the field as divided into two or perhaps three major subfields, each with different goals and objectives, with different methodologies, and with different types of standards by which they are judged.

The basic division, recently adopted in the major AI conferences, is between AI as a science and AI as engineering. Finer divisions are also suggested. Thus, Bundy, for example, advocates a threefold subdivision of AI [2, 3].

1. Mainstream AI: the investigation of computational techniques that have the potential for exhibiting intelligent behavior.

2. Technological AI: the application of these computational techniques to the building of commercial products, including knowledge engineering and intelligent robotics.
3. Scientific AI: the application of these computational techniques to model human cognition, that is, most scientific AI is cognitive science.

ILL STRUCTURE IS CHARACTERISTIC OF AI

A concept like AI, or even the word "intelligence," which cannot usefully be pinned down with mathematical precision, is an ill-structured concept. Ill-structured concepts are part and parcel of what makes AI different from, say, software engineering.

Consider the following old linguistic chestnut, which makes the point with a minimum of fuss: Is it correct for a computer to translate, "The spirit is willing but the flesh is weak," into Russian as, "The vodka is good but the meat stinks"? Probably not, but it might be adequate; it depends on the context. Context sensitivity is an important characteristic of intelligence and, therefore, of AI as well.

ADEQUACY: THE CRITERION OF AI

What is the correct translation? Is there a single correct translation? Is a translation, based on the most probable (in some sense) implications, the correct answer? Deciding on the correct translation, or if indeed there is a correct translation, is another context-dependent problem. Ill-structured concepts abound in the everyday lives of all of us. A consequence is that correctness is not a useful measure to apply; the criterion of AI is adequacy.

So what should our intelligent, natural language translation system produce if it is to be judged as adequate? The assessment of adequacy is a difficult task. If the system performs a useful role in a manner that is satisfactory to the persons making use of it, then that system is adequate. Adequacy judgments can be viewed in terms of absence of inadequacy; If the system exhibits no major inadequacies when performing a useful task, then it is an adequate system. Assessment of the adequacy (and even more difficult, relative adequacy) of complex AI systems is currently one of the difficult, unsolved problems in AI.

CONTEXT SENSITIVITY IS TYPICAL OF AI

The design of a natural language interface, which will afford computer users the convenience of communicating with a computer system in English rather than a programming language, requires that we first construct natural language understanding (NLU) systems. But the meaning of a sentence may depend more on who said it, why they said it, and when they said it than the actual words used. The NLU problem is highly context sensitive; it does not appear to be readily decomposable into a collection

of relatively independent, or context-free, modules. Tightly coupled context sensitivity is typical of AI problems.

Hewitt uses the term "open systems" to describe much the same feature of AI [4]. "Open systems," he says, "are always subject to communications and constraints from outside."

THE TURING TEST

Breadth and depth of communication abilities seem to be strong indicators of intelligence. At the genesis of modern AI, in 1950, the British mathematician Alan Turing set up the paradigm for deciding the question, Can machines think? Turing's imitation game gives us the breadth and depth of conversational abilities as the probe for evaluating the intelligence of a computer system [5]. Despite many years of criticism, the basic paradigm has endured: In essence, if in extended and unconstrained natural language communication a computer system is indistinguishable from a human being, then that system possesses intelligence. Hence, we have a definition, of sorts, of AI.

Turing's choice of natural language as the means of scrutinizing a system for intelligence turns out to be quite apt. The underlying implication is that if a system is capable of conversing sensibly in prolonged and unrestricted discourse, then the system is also capable of the other many and varied accoutrements of intelligence. The validity of the implication is a subject for endless debate.

But, as we shall see, natural language is so persuasive an indicator of intelligence that in AI the test backfires on us repeatedly.

THE ELIZA SYNDROME

Since 1950 there have been some weak claims (usually not by the system designers themselves) that some programs have passed the Turing test. The most famous of these programs is ELIZA [6]. Numerous introductions to AI offer a short and sensible dialogue between a human and the ELIZA program. A commonly encountered example of an ELIZA-human dialogue is the following (ELIZA's utterances are in capitals):

Men are all alike.

IN WHAT WAY

They're always bugging us about something or other.

CAN YOU THINK OF A SPECIFIC EXAMPLE

Well, my boyfriend made me come here.

YOUR BOYFRIEND MADE YOU COME HERE

He says I'm depressed much of the time.

I AM SORRY TO HEAR THAT YOU ARE DEPRESSED

It's true. I am unhappy.

DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY

I need some help; that much seems certain.

WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP

Perhaps I could learn to get along with my mother.

TELL ME MORE ABOUT YOUR FAMILY

My mother takes care of me.

WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU

My father.

YOUR FATHER

You are like my father, in some ways.

WHAT RESEMBLANCE DO YOU SEE

You are not very aggressive but I think you don't want me to notice that.

WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE

You don't argue with me.

WHY DO YOU THINK I DON'T ARGUE WITH YOU

You are afraid of me.

DOES IT PLEASE YOU TO THINK I AM AFRAID OF YOU

My father is afraid of everybody.

WHAT ELSE COMES TO YOUR MIND WHEN YOU THINK OF YOUR FATHER

Bullies.

DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR

BOYFRIEND MADE YOU COME HERE

And this persuasive snippet is supposed to buttress the claim that Turing's test is about to be passed.

It is true that certain AI programs are capable of an extended and sensible dialogue in English, and, moreover, it is a dialogue that carries significant information in both directions. A number of expert systems, such as the TEIRESIAS-MYCIN combination (a natural language interface to an expert system [7]), fall into this category. But the range of sensible dialogue is extremely narrow and fragile. A fundamental feature of Turing's test is that the dialogue should be broad ranging and robust.

The ELIZA program fails the Turing test dismally on two critical counts: Sensible dialogue is possible only for short periods and within very narrow constraints.

Joseph Weizenbaum, ELIZA's creator, was motivated to write a book just to counter the adverse effect that his program was having on casual computer users [8]. A disturbingly large range of people insisted on investing the ELIZA program with intelligence despite all protestations to the contrary by Weizenbaum himself.

Thus, we have the ELIZA syndrome to guard against in AI: Seemingly intelligent snippets of conversational dialogue or of problem-solving ability are an almost irresistible basis for unwarranted grandiose assumptions as to underlying abilities and, hence, potential domains of application. The results of applying underpowered, or worse, totally inappropriate programs to societal problems could be catastrophic. I have discussed elsewhere the societal problems that may be engendered by "almost-AI" programs [9].

THE MYTHOLOGY OF AI

The Academic Myth: Intellectualism is the Peak of Intelligence

If we can simulate man's difficult intellectual achievements, like chess playing and theorem proving, then solutions to the simpler problems, like cleaning a house or working on a production line, will fall out along the way. Wrong! We are tackling the intellectually impressive, but not the harder problems, first.

Starting with the easiest examples is, of course, a legitimate way to proceed provided we do not present such examples as the ultimate goal of AI. Far from being the major peak of intelligence, intellectualism may be only one of the minor foothills. Furthermore, it is a moot point as to whether any path up this foothill leads to the major peaks. A charge often leveled at AI researchers is that they are climbing trees as a first step toward reaching the moon.

In an essay on the influence of automation on society, Simon records the surprise of AI researchers when they realized that the intellectual aspects of intelligence were the easiest to simulate with a computer [10].

A wealth of knowledge is the basis of successful adaptive behavior, and such behavior is the key to solving these ill-structured problems adequately. Traditionally, we do not marvel at the knowledge-guided problem-solving abilities of a laborer or that other much maligned super problem solver, the housewife.

It is an indisputable fact that the cleaning of a house requires a larger, more diverse, and complex knowledge base than does the playing of international standard chess. Yet success in the latter pursuit is typically taken to be indicative of intelligence, whereas the former is related as a menial chore. Housecleaning is in fact a highly complex ill-structured problem. The first general housecleaning robots are a very long way off.

The Spock Myth: Artificial Intelligence Will Not Be Impeded by Human Weaknesses such as Emotions and Nonlogical Reasoning

It is a common misconception that the cold, totally rational automaton will put us to shame as it slices logically through the problems of everyday life. Star Trek's favorite Vulcan is mythical in more than one sense.

Is human intelligence a local optimum, given the conditions of life on earth? Or is it an evolutionary back alley, perhaps a dead end? Is machine intelligence going to be the ruthless competitor dominating evolution's best effort as placental mammals quickly overran their marsupial counterparts in Australia? After several decades of feverish activity in AI research, the answer is still not known; it is a question that only the passage of time will really answer, although there are a number of reasons to believe that evolution came close to the optimum with *Homo sapiens* [11]. This argument implies that human intelligence with all of its obvious weaknesses may be an optimum (within the constraints of the empirical world); it may be an upper bound on the possibilities for AI.

Hofstadter suggests that human foibles, such as love and hate, might simply emerge as epiphenomena in any sufficiently complex system [12]. Even if there was no attempt to program love, for example, into a system, it might arise once the system reached a certain level of complexity, that is, we might be prepared to admit that the system appeared to act lovingly as a result of observing its behavior. So, although an AI system designer would not purposely include the problematic features of human intelligence within a system, these features may just emerge as necessary side effects of intelligent behavior. Much as workers in AI would like to extract the *sapiens* without the *homo* (like a wrinkle from its shell), it is not at all obvious that the two are really separable.

THE GOALS OF AI RESEARCH

We can distinguish different levels of approach to the category described earlier as scientific AI. Two such levels are one in which the theoretical primitives are the functional units of classical psychology—the cognitive science level—and one in which the primitive elements can be directly related to brain physiology—the brain modeling level.

The cognitive science branch of AI seeks to throw light upon the nature of human intelligence, its scope and limitations, and how it works. Within this domain, the computer is a tool for modeling theories of human intelligence. The cognitive scientist theorizes in terms of high-level concepts: long-term and short-term memory, semantic mismatch detectors, semantic networks, and so forth. Such components have no obvious correspondence with anatomical features of the brain; they are high-level abstractions rather closer to the behavioral characteristics that they seek to explain than to the organ from which the behavior (presumably) arises. Most theoretical AI falls into this subcategory, as a perusal of Barr and Feigenbaum [13] will confirm.

By way of contrast, the brain-modeling approach to AI theorizes and models in terms of units that can be related to brain structure: cell assemblies, neural networks, even individual neurons. The basic building blocks for the theory may now be directly observable, but piecing them together to explain observed behaviors represents a formidable problem. Consider explaining the details of a weather pattern in terms of intermolecular interactions rather than temperature, pressure, prevailing winds, and so forth (see, e.g., Refs. 14 and 15; Ref. 16 gives an analysis of one mechanism from a number of different levels).

But much of AI research does not ostensibly seek to answer any questions at all about human intelligence—roughly, mainstream AI under the previous categorization. The goal of this work is to emulate or surpass human performance of certain intelligent activities. The strongest link with human intelligence is that it represents a performance measure to strive for and then surpass.

This is also the domain of major practical success; it is the source of technological AI. Thus, expert systems, it is claimed, can surpass human experts using, it seems, totally different mechanisms. But I would claim [17] that despite the almost standard disclaimer to the contrary, some degree of structural correspondence between AI modules and hypothesized

mechanisms of human intelligence is both a perceivable and a necessary feature of this work, especially when a practical application is envisaged. Comprehensibility is at a premium (because the maintainability of the software product relies upon understanding the program), and reasoning by analogy with one's own supposed thought processes is a necessary vehicle for realizing this goal. In the expert systems mentioned above, for example, automatic explanation of system's diagnoses (or whatever) in psychologically meaningful terms, rather than in terms of the implementation structures, is seen as an important problem. As a further and more specific example, Michie calls for a technique of "structured induction" to combat the inscrutability to man of the products of machine learning [18].

Cultural differences all too frequently block and distort mutual communication and understanding between instances of natural intelligence. Communication difficulties are a problem within the relatively homogeneous umbrella of human intelligence, and we must beware of the potential for escalation of this problem when we introduce an AI.

Next, we must consider heuristics, a key concept in the realization of AI. A good heuristic is any strategy that works, and it is not explicitly linked to the mechanisms that humans apparently use. So, at first glance the use of heuristics appears to deny the humanness claimed in the preceding paragraphs, but, as you will see, the ultimate source of heuristics is inevitably human. The result is that an AI system is bound to display a certain humanness. (But see the section on machine learning, which encompasses the potential for developing nonhuman heuristics.)

THE HEURISTIC APPROACH

If it works, use it, is the principle behind heuristic programming. A heuristic is a rule of thumb: a procedure that achieves a certain goal on an acceptable proportion of occasions. A heuristic to guide machine learning might be the following: *A machine should learn only significant events.* To incorporate a heuristic within a program, we generate an algorithm to approximate it—a heuristic algorithm, in short. So an implementation of the above heuristic must include explicit code to approximate the following: *What makes an event significant, what aspects of a significant event will be learned, and so forth?*

A house cleaning robot will require a heuristic, such as *water plants when dry.* This appears to be a reasonable rule that will be appropriate most of the time with some notable exceptions, that is, cacti and artificial plants. However, although this statement of the heuristic would probably be sufficient if you were instructing a friend who was taking care of your house for a while, it will need to be transformed into a precise, algorithmic rule for the housecleaning robot.

Let us try for a more precise statement: *Add water, in an amount appropriate for the size of the plant, to every plant whose soil has a relative moisture content less than some prespecified value.* This formulation would seem to explain what is wanted—or does it? Following is a list of some of the questions that would have to be answered before this heuristic attains the algorithmic specificity currently necessary in AI systems.

- What does "appropriate for" mean? Proportional to? If so, proportional to what? Size of plant?
- How do you measure size? Height; estimated weight; surface area of soil; volume of soil?
- "Relative moisture content" Relative to what? Size of plant; volume of soil; surface area of soil; depth of soil?
- Where do you measure the moisture content? On soil surface; x inches below the surface; the average of several arbitrary probes?

The questions are many, reasonable answers are legion, and correct answers are exceptional. But the questions must be paired with precise answers before we will obtain a machine-executable version of the original heuristic. The selection of a particular set of necessarily somewhat arbitrary answers will result in a particular algorithmic approximation to our original heuristic. There are very many such algorithmic approximations and very few ways to choose between them.

So a heuristic implemented in an AI program is more accurately viewed as one of many possible algorithmic approximations to the heuristic, but this is an exceedingly clumsy and pedantic phrase so I will use the terms heuristic and heuristic algorithm instead.

An important question here is Why is it that I can tell my friend to water the plants when dry but I must make a lot of extra decisions before I can tell my robot much the same thing, especially if, as I claim, there are no good reasons to believe that AI will suffer from any limitations that do not afflict human intelligence?

The difference is just a question of time and is due to the vast amount of knowledge that my friend can apply to understanding the heuristic as opposed to the paucity of such knowledge that a robot will have in its knowledge base in the foreseeable future. My friend will (unconsciously) generate his/her own algorithmic approximation to the heuristic; it will be many years before we can trust a robot to do the same.

I can reasonably expect my friend's approximation to be acceptable to me because of the overall similarity between his/her applicable knowledge and mine, but I cannot be sure of it. Heuristics come with no guarantees. Then, why do we use heuristics? Wouldn't some proven algorithm be better?

Proven algorithms would certainly be preferable, but they are unfortunately either hard to find or hopelessly inefficient in AI. The use of a heuristic represents the best known solution procedure that will succeed, if it does succeed, within reasonable time and space constraints.

The space/time trade-offs here are not just microseconds, or even days or weeks. They may well run to centuries or the lifetime of the universe! The search for efficient heuristics is well-motivated when the only known proven algorithms will take years to terminate. People just won't wait that long.

The game of chess provides a simple example. In principle, the best move to be made from a given board configuration is easily computed. We consider all the possible moves, all the possible replies by the opponent, and so on, until we have generated the full game tree down to all of the terminal win, lose, or draw situations. Then a minimax algorithm is applied to specify correctly the best move to make. Unfortunately, both the space and time resources required to use this algorithm are impossible to meet.

The full game tree for an arbitrary board configuration in chess will generally more than overfill all existing computer memories. The time required for generating and processing it is of the order of the time that the universe has been in existence. Clearly a heuristic is called for. In practice, heuristics are used to estimate the relative goodness or badness of potential moves without generating the full game tree.

Pearl examines many heuristics in the relatively well-defined but, nevertheless, rich AI domain of games and puzzles [19]. He states, "It is the nature of good heuristics both that they provide a simple means of indicating which among several courses of action is to be preferred, and that they are not necessarily guaranteed to identify the most effective course of action, but do so sufficiently often."

The guarantees, or lack of them, are one thing, but where does the heuristic come from in the first place? The sources are varied and not well understood. Some heuristics are definitely plucked from the air (no different from many algorithms). A common source of inspiration for heuristics is, naturally, oneself. Introspection is a very convenient strategy, but it must be used with caution.

To return to the chess example: I can design a first approximation to a best-move heuristic by introspection and rationalization of the strategies that I appear to use when playing chess. Typically, the heuristic would be composed of components that compute first approximations to king threat, piece mobility, and so forth. But if you are not an expert yourself, then a better strategy might be to attempt to extract heuristics from an expert chess player.

Recognized experts are indeed a good source of heuristics. Unfortunately, experts are no more expert at introspection than you or I. All indications are that they are full of expertise and replete with high-quality heuristics, but how to pour them out of the expert and into a program is a problem that is receiving a lot of attention. Later, in the context of machine learning, I will describe several approaches to this problem of transfer of expertise.

Provided that the first guess for a heuristic strategy focuses upon the significant dimensions of the problem domain, there is every chance that the heuristic can be tuned to adequacy by means of an iterative process. This tuning process may be totally automatic, in which case it is described in the section on machine learning, or it may be, in essence, the run-debug-edit paradigm, a man-machine symbiosis and, as such, it is considered in detail in the section on AI methodology.

THE EARLY DAYS—GRAND PLANS

Many early efforts were directed at broadly based, general tasks, such as machine translation from, for example, English to Russian [20]. Another worldwide set of projects was bent on producing mobile, perceptive robots, such as Stanford Institute's Shakey [21]. It quickly became apparent that such broad targets were far too difficult (easy to hit but hard to kill). AI research is now closely focused onto much more specific problems, such as natural language discourse in very limited domains or hand-eye coordination in very limited settings.

Game playing is one area that attracted early attention and proved susceptible to machine implementation. Games are a group of well-specified and constrained, but still challenging, intellectual problems. Moreover, it is a domain in which the performance measure can be applied accurately: A chess-playing program can be ranked accurately (as accurately as humans are ranked) by its win, lose, or draw performances against ranked human chess players (or even other chess-playing programs). The current level of the best chess-playing programs is that of a national champion.

THE SAMUEL PHENOMENON

Samuel's checker-playing program was an early landmark study in AI [22]. This program beat the world champion and Samuel himself, and this achievement is almost as remarkable as it appears. Samuel designed a program that played the game legally, and he also included mechanisms that stored and reused successful moves. The program improved its performance with experience—it learned.

Samuel's program learned by both rote and generalization. The rote learning involved storing both expert moves and board configurations that it had evaluated for itself. The generalization learning was implemented as adjusting the weights associated with selected characteristics, such as king advantage and piece mobility, in an attempt to discover combinations that produced better play (as judged by both human experts and by direct competition against another copy of the program with different weightings).

The world champion was caught off guard in the first match, but then he analyzed the program's basic strategies and subsequently beat it easily. Although the program learned from experience, the learning was of a rudimentary and fixed nature; it did not learn new strategies of play.

The most significant achievement was that it could beat its designer consistently. Samuel has no pretensions to checker expertise; nevertheless, he did write the program and the program trounced him—shades of the robot that refuses to obey its creator? I think so. It definitely throws doubt into the old adage, Computers can only do what you tell them to do. In a weak sense, this statement is true, but it has several significant flaws when used as an assertion of the inherent limitations of AI. Samuel did not tell the computer to beat him; he told it in a quite straightforward way how to play legal checkers and how to improve its performance as a result of experience.

Now, we also have the Samuel phenomenon in AI. With complex programs, especially if they contain learning mechanisms, it is, in practice, impossible to predict the full range of the program's possible behavior. More succinctly, a programmer quickly loses track of exactly what he/she has told the program to do, or, rather, the consequences of what he/she has told it to do.

"Computers are literal-minded brutes and do exactly what they are told; no more and no less." Remember the Samuel phenomenon and treat such statements with caution.

EXPERT SYSTEMS AND KNOWLEDGE BASES

Significant successes have also been achieved within less well-structured but very specialized domains: DENDRAL [23], a program that analyzes chemical mass spectrograms, and MYCIN [24], a medical diagnosis program, are just two examples. These programs, which can match and even outperform highly trained human experts, are known as expert systems.

The following example (reprinted in Ref. 13, p. 238) is an excerpt from a session with a South American geography tutor. The student's questions and answers are preceded by two asterisks (**), and annotations are capitalized.

What is the language of Argentina?

***Probably Spanish.*

Very good.

Approximately what is the area of Argentina?

STUDENT CHANGES TOPIC

***Tell me something about Peru.*

Peru is a country.

It is located in South America.

The capital of Peru is Lima.

Now, answer the question you didn't answer before.

FINALLY ANSWERS THE QUESTION

***The area of Argentina is 1,100,000 square miles.*

Approximately true, but a more exact value is 1,200,000.

The eastern bordering country of Peru is...

STUDENT MAKES AN ERROR AND IS CORRECTED

***Catholicism.*

Wrong. Catholicism is not a country but a religion.

The correct answer is Brazil.

Where is Peru?

STUDENT CHANGES TOPIC AGAIN

***Is Asuncion hot?*

Seeing that Asuncion is part of Paraguay, yes.

***What is the most important difference between Peru and Paraguay?*

The primary difference is the population of Peru is larger by a factor of 7.8.

Such dialogues are definitely suggestive of intelligence (but remember the ELIZA syndrome), and a handful of expert systems have undergone field trials that appear to demonstrate that adequate or better performances are indeed reproducible. But note that the successful expert systems do

not deal with natural language primarily; they operate in domains of abstract, technical expertise.

The success of expert systems rests largely on the very restricted and specialized nature of the domains in which they operate. Intelligent behavior within the ill-defined empirical world (as opposed to well-defined game-playing and puzzle-solving situations) is founded upon vast amounts of information.

A well-known computer science trade-off is between the complexity of the algorithm and the data structure. A given problem may be solved with a simple algorithm operating on a complex data structure or with a complex algorithm operating on a simple data structure. In AI, as usual, the problem is worse. Both the algorithm and the data structure tend to hover on the threshold of human comprehension. It is then crucial to find knowledge representation schemes that can store large amounts of information, be readily modifiable, allow fast, flexible access, and that can be processed without need for unduly complex algorithms—a tall order, but one that is considered to be of paramount importance in AI and constitutes a major subfield; knowledge representation.

KNOWLEDGE REPRESENTATION

The following simple example illustrates a popular approach to knowledge representation in a "LISPese" notation (more on LISP later).

```
(F1 (HOBBIT BILBO))
(F2 (UNCLEOF FRODO BILBO))
(F1 (IF THEN (UNCLEOF X Y) (NEPHEWOF Y X)))
(R2 (IF THEN (AND (UNCLEOF X Y) (HOBBIT Y)) (HOBBIT X)))
```

Four entries constitute this knowledge base: two facts, F1 and F2, and two rules, R1 and R2. The two facts are the given truths of the system, and in combination with the rules they allow us to infer further true statements. Thus, if we take R1 and instantiate X as FRODO and Y as BILBO, the truth of the resultant condition (UNCLEOF FRODO BILBO), given by F2, enables us to infer the conclusion, to wit, (NEPHEWOF BILBO FRODO), and so on.

Specific knowledge may be stored as facts and general knowledge may be stored as rules, and by combining the two we can infer many other facts.

Having provided an example of knowledge representation, I will briefly survey this very important subfield of AI—the area of knowledge representation.

The example given above is one of a logical representation of knowledge. This class of representation has the advantage that we can employ the well-defined semantics of logic. In addition, the knowledge base is a fairly simple collection of readily understandable units.

But the fact that such knowledge bases are, more or less, a homogeneous collection of simple units results in an overall "flatness," a lack of perceptible higher-level structure and, hence, a lack of comprehensibility. In

addition, the well-defined nature of the basic logical operations is obtained at the cost of severe limitations on applicability in representations of the empirical world. Some problematic areas are incremental acquisition of knowledge, beliefs about truth rather than truth in some absolute sense, and the combination of general assumptions with exceptional situations that may override the normal assumption.

All of these characteristics of empirical knowledge and its use to support intelligent behavior can be reduced to the necessity to be able to add new knowledge that may then alter the truth of some current knowledge. In logical terms this leads us into a less well-understood logical world—the domain of nonmonotonic logic.

Thus, in the example given above, an initial belief (as opposed to an absolute axiomatic truth) that (HOBBIT BILBO) is true might have to be abandoned in the light of subsequent evidence. In general, this sort of change is likely to have repercussions throughout the knowledge base; it is this context sensitivity that (as we know) is typical of AI problems and thus undermines the utility of these simple logical representations of knowledge.

From an early, enthusiastic commitment to logical schemes based on the observation that some reasoning is logic-based, there was subsequent disenchantment and rejection due to the realization that much of the information processing necessary to support intelligent behavior did not appear to be logic based at all—at least not based on any simple and well-understood logic. In an appendix to an essay on knowledge representation, Minsky explains why he thinks that "logical" approaches will not work [25]. One of his points of argument is that logical axioms are "permissive," that is, "Each axiom added means more theorems: None can disappear." Hewitt challenges the sufficiency of current logic-based techniques for developing the intelligent systems of the future [4].

Nevertheless, current interest has turned somewhat back to logical schemes in two ways: First, there are efforts to construct more powerful logics, and second, there is interest in hybrid schemes that partition representations of knowledge so that logic is used where it is appropriate and other schemes are used where logic does not seem to be appropriate—a high-level modularization of the knowledge representation problem.

An important problem with this logical type of scheme is that even if a logically valid procedure, such as deduction, is possible in principle for some application, it may not be possible in practice. The practical impossibility is a result of the fact that the logical mechanisms are unguided and the number of search paths then quickly becomes the limiting factor.

Using the preceding knowledge base, we can deduce the truth of (HOB-BIT FRODO). This involves using both F1 and F2 to allow us to draw the required inference from R2. However, it could have involved any other sequence of the knowledge-base elements—the algorithm must simply keep trying all possibilities until it succeeds (if it can succeed). This unguided, brute-force approach quickly becomes too time consuming for all but the most trivial of knowledge bases.

Consider the original knowledge base with one extra rule, R3, which says that if X is the NEPHEW OF Y, then Y is the UNCLE OF X. That would appear to be a true and innocuous piece of information to add.

```

(F1 (HOBBIT BILBO))
(F2 (UNCLEOF FRODO BILBO))
(R1 (IFTHEN (UNCLEOF X Y) (NEPHEWOF Y X)))
(R2 (IFTHEN (AND (UNCLEOF X Y) (HOBBIT Y)) (HOBBIT X))
(R3 (IFTHEN (NEPHEWOF Y X) (UNCLEOF X Y)))

```

Now if we want to examine the truth of (HOBBIT FRODO), we can deduce from F2 and R1 that (NEPHEWOF BILBO FRODO) is true. This is not what we want, so we use, say R3; then we know (UNCLEOF FRODO BILBO) is true, but this is not what we want either, so we use R1, and so forth. It is quite possible to follow endless loops of deductions that are not at all obvious in more complex situations. There is generally no guarantee that attempted proofs will terminate. (Turing and his halting problem haunt us even here.)

So, even within their limited domain, automatic proving mechanisms require guidance before they become viable schemes. Two general control strategies are forward and backward chaining. In a forward chaining strategy, we start with our knowledge base and generate implications from it hoping to chance upon the particular one that we are interested in proving.

The opposite strategy is somewhat more guided (it could hardly be less). We start with what we want to prove (e.g., (HOBBIT FRODO)) and determine what would have to be true in order to prove this goal. With the earlier knowledge base, we can see that our goal will be the conclusion of R2 if we replace X by FRODO. Thus, we now need to determine the truth of the condition of R2 with this replacement. We must now prove both (UNCLEOF FRODO Y) for some value of Y and (HOBBIT Y) for the same value of Y. F2 and F1 give us the proofs we need when Y is replaced by BILBO. Hence, we proved the original theorem by chaining backward from it.

Other somewhat directed proof strategies are available, for example, the resolution principle, but the amount of guidance is still minimal and insufficient for many practical applications. Much more guidance is required.

The solution is to use sophisticated heuristic control strategies that can exploit both the general context of a situation and the specifics of each individual attempted proof. This problem is then far from solved.

Having reached this point, I will briefly consider the two other major classes of knowledge representation schemes and, at the same time, also continue with some more general problems of Knowledge representation and use.

Mylopoulos and Levesque provide a concise and informative overview of knowledge representation [26]. They characterize the other two classes of schemes as procedural and network representations.

In a procedural scheme, knowledge is represented as a set of processes. One or more processes is activated by certain states, and the execution of the activated process(es) transforms the current state into a new one, and so on.

Production systems are the one type of procedural knowledge representation. The earlier logical knowledge base might be represented by the following productions:

```

R1 IF (UNCLEOF X Y) THEN (NEPHEWOF Y X)
R2 IF (UNCLEOF X Y) AND (HOBBIT Y) THEN (HOBBIT X)

```


and the current state, which is given by the facts,

F1 (HOBBIT BILBO)

F2 (UNCLEOF FRODO BILBO)

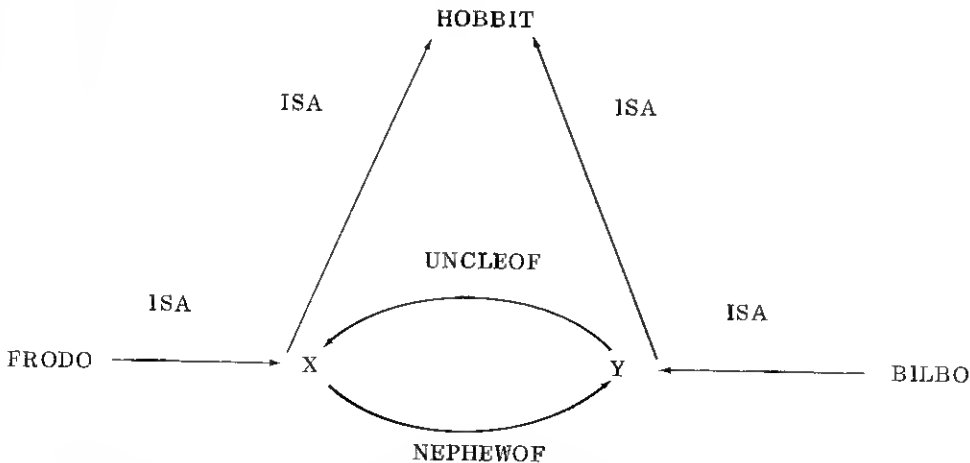
Notice that the facts will satisfy the condition part of both rules; thus, both rules may be executed (or "fire") in this state. If we impose a restriction that only one rule may fire for any state, a decision must then be made between these two—the conflict must be resolved. Again, we need control information, but this time it can be built into the set of production rules as well as imposed by external heuristic strategies.

If we assume that the set of production rules is scanned sequentially from top to bottom until the first rule that can fire is encountered, then R1 will be fired given the current facts. Control information is already implicit in the ordering of the rules. Actually, rule R2 will never be fired because any state that satisfies the condition part of R2 will also satisfy the condition part of R1, and thus R1 will always be fired in preference to R2.

Building control information into the set of production rules can eliminate much of the searching that plagues logical schemes. But the introduction of these context dependencies does, of course, aggravate the problem of understanding and modifying production systems.

The general trade-off here is that between searching context-free representations and providing guidance at the cost of introducing context dependencies.

The third type of scheme represents knowledge as a collection of "objects" and relationships between them, that is, a network representation, often called semantic networks. A semantic network representation of our knowledge base might be



The above representation "might be" a reasonable interpretation of the original logical representation because the semantic net representation does not define much (anything?). This is the major drawback of such represen-

tations; there are no well-defined formal semantics for network representations.

I think the network that I have drawn looks like a plausible rendering of the original representation, but how to interpret it in any well-defined sense is not defined.

One difference between the logical and the network representations that I have provided is that the latter is symmetrical and the former is not. I can demonstrate this lack of symmetry about the HOBBITS in the logical scheme by substituting FRODO for BILBO in F1 of the original representation and then trying to prove (HOBBIT BILBO)—it cannot be done. F1 appears to be redundant in the semantic network; the fact that BILBO and FRODO are both HOBBITS appears to be an implication of F2, R1, and R2 in the semantic network representation.

However, in the semantic network, BILBO and FRODO are symmetrical with the exception of the complementary relations UNCLEOF and NEPHEWOF.

This introduction of symmetry is just one manifestation of the fact that the logical representation is a collection of context-free units, whereas the semantic network is a single unit—information must be added in order to generate a cohesive whole.

In particular, the logical representation specifies that the UNCLEOF relation implies NEPHEWOF but not vice versa—the semantic network includes the complementary relation, apparently. In addition, the graphical representation is a considerable aid to understanding, in a general sense, the knowledge structure being represented.

But all three representational schemes, semantic nets in particular, have a tendency to induce the ELIZA syndrome in unsuspecting onlookers. You might, for example, be inclined to view my knowledge representations as somewhat male chauvinistic. Nothing could be further from the truth (or my mind)!

Surely, the axiom that says UNCLEOF implies NEPHEWOF with no mention of NIECEOF is indicative of a bias in favor of the XY chromosome combination. And Bilbo and Frodo were male Hobbits, weren't they? No and yes are the answers, in that order. The relationship UNCLEOF has no necessary connection with the status of a parent's male sibling, and Bilbo is not necessarily BILBO.

There is no mention of sex in any form in my knowledge bases. In fact, there is no mention of anything semantic (apart from the formal semantics of, e.g., logic). Each of the well-defined processes described has been a syntactic activity—formal manipulations of abstract patterns that were arbitrarily labeled. It is in the reading of everyday semantic interpretations into these labels that the ELIZA syndrome strikes.

Woods addresses the general problem of what the notations and structures used in semantic networks can mean and with the need for an explicit understanding of the intended meaning for various types of links and arcs [27]. He also examines the representational adequacy of semantic networks for knowledge representation and finds them lacking in a number of important respects.

The problem of the unfounded persuasiveness of semantic networks is another example of the form-content problem. The form of semantic networks constitutes a difficult-to-resist temptation to the human perceptual

mechanism. We invariably assume that a rich content underlies the representation, and the richness of our assumptions is rarely warranted by the semantics actually implemented.

Elsewhere [28], I have previously drawn attention to the syntax-semantics (or form-content) confusion that surrounds semantic network representations. I have argued that syntax and semantics are neither absolute nor clearly separable concepts; this mixing and lack of absolute definitions are, I claim, major reasons why semantic networks invariably infect the casual observer with the ELIZA syndrome.

In a searching examination of knowledge representation, Hayes addresses this problem in terms of the "fidelity" of a formalization [29]. "It is perilously easy to think that one's formalization has captured a concept (e.g., because one has used a convincing-sounding token to stand for it), when in fact, for all the formalization knows, the token might denote something altogether more elementary."

A closely related criticism of most current knowledge bases is that the knowledge represented does not embody a deep understanding of the domain; instead, it is a collection of "pattern \rightarrow action" rules—a representation of superficial knowledge that excludes the possibility of using such knowledge to solve hard problems. The underlying principles are absent and so cannot be used to support deep reasoning.

Chandrasekaran and Mittal provide a lucid examination of this contention, which they support on the whole [30]. But they do stress that despite the many calls for more deep knowledge, there is no general agreement on the form and content of these deeper structures. In particular, they illustrate that the popular belief, that "causal" knowledge is deep knowledge, does not stand up to close scrutiny. In a nutshell, substituting "pattern CAUSES action," for "pattern \rightarrow action" is no guarantee that deeper knowledge has been encoded.

Let us deal with one last and important question concerning representations of knowledge in general.

Suppose you are told that the computer in front of you contains an expert system—an expert on "Middle earth" (i.e., you are given a context within which to draw semantic implications such as BILBO is Bilbo). Digging deeply into your own knowledge base (the one in your head), you retrieve something about little people named Hobbits, and a question for the expert systems springs into mind. You type the query

(HOBBIT HOLLY)

What is the correct response, given that the expert system only has our original, ludicrously small, knowledge base with which to work?

Clearly the response should be that this statement is false (Holly is a Hobbie, not a Hobbit), and the system would soon fail to deduce the truth of this query; thus, it would correctly output

No, Holly is not a Hobbit.

But I have glossed over an important implication here; that failure to prove truth implies falsity. There is nothing in the knowledge base that

allows us to explicitly conclude that Holly is not a hobbit, that is, no instance or possibility of deducing NOT (HOBBIT HOLLY).

Under the same rules the queries (WIZARD GANDALF) and (BADGUYS GOBLINS) would turn out to be false. And we know that they are true, at least in Middle earth—and there is the crux of the problem. As Levesque says, "The user of the knowledge base must distinguish between what is known by the knowledge base and what is true in the intended application area" [31]. Knowledge bases tend to be incomplete, that is, they often do not have all the information necessary to answer a query.

Incompleteness of knowledge bases is a property that we must be able to cope with in the ill-defined and dynamic domains of AI. A common way to deal with this incompleteness problem is to use the so-called "closed world assumption": The knowledge base is assumed to know everything about its world; thus, if it doesn't know that something is true, that something is then false.

Unfortunately, this straightforward solution to the incompleteness problem is not very satisfactory in some AI applications; the system needs to know about what it doesn't know (as a guide to knowledge acquisition, e.g.,) as distinct from knowing that certain things are false.

Here we see a need for metaknowledge arising. As usual, special-purpose heuristics have been the route to dealing with this general problem. Collins et al. describe a selection of such strategies [32] and illustrate their choices with SCHOLAR [33], the geography tutoring system that I used as an example earlier.

The beginnings of a generalized and formal approach to these problems of incomplete knowledge bases have been described by Levesque [31]. He is developing a formal language within which one can query a knowledge base as to what it knows and does not know. Thus, in our earlier example the metalevel query, "Do you know that Holly is a Hobbit?" would be answered, "I do not know that Holly is a Hobbit." Neither (HOBBIT HOLLY) nor NOT(HOBBIT HOLLY) can be proved within the current knowledge base.

I should emphasize that although Levesque's work is a promising approach to the possibility for formalizing metaknowledge, it is ongoing research, and a number of important problems must be solved before it can become a practically useful technique.

When we add the knowledge base processing algorithms to the representation structure, we then have the core of the implementation aspect of knowledge engineering (of course, it is not really possible to try to separate the algorithm from the data structures; certain aspects of the former define the latter). If we take most of the vagaries and ill structure of everyday life out of the picture, we are left with realistic and serious but potentially tractable AI problems. These problems are the grist for the expert system mill.

They are also some of the best candidates for AI in software engineering, so we might be well advised to spend more time on the practical feasibility and potential scope and limitations of a scheme for packaging general intelligence into small, self-contained, highly limited intelligent chunks. But first, let us examine the machine-executable notations that are available to us for implementing AI software.

AI PROGRAMMING LANGUAGES

Along with most other subfields of computer science, AI has not been lax in its efforts to invent new programming languages to solve its own particular brand of problems. An AI programming language must support flexible, dynamic systems. Both data structures and algorithms should be modifiable at run time. In fact, this convenient and useful division—data structure and algorithm—becomes a blurred line in AI: Data structures need to be executed, and algorithms need to be manipulated as though they are data structures.

One of the early high-level programming languages, LISP [34], is still very much the vernacular of AI, although AI aficionados actually work with the extended dialects, such as MACLISP [35] or INTERLISP [36]—dialects plus extras that constitute two of the finest program development environments available. An earlier but comprehensive description of "the LISP experience" is given by Sandewall [37].

As an example of a LISP program, we can specify a function that sums a list of values, for example, integers, and returns that sum as its result. Consider the following pseudocode specification of such a function:

```
function LISTSUM,
  IF the list contains no values
    THEN return zero as the function value
  ELSE return the first list value added to
    the sum of the rest of the list values as
    the function value
```

The only slightly disturbing feature of this algorithm, especially to the nonrecursive programmer, is the hint of circularity in the ELSE clause. The sum of a nonempty list is obtained by adding the value of the first element to the sum of the rest of the list. There are two reasons that, together, save the algorithm from endless circularity: (a) The rest of the list (i.e., the list without its first element) is always shorter than the list it derives from, and (b) the THEN clause returns a result for the shortest possible list (i.e., the empty list).

Thus, it is true that there is a circularity within the ELSE clause: The computation of the sum of a list of values is expressed in terms of the sum of a shorter list of values. But this circular, self-referencing, known as recursion, will eventually terminate when the rest of the list is empty; the function will then "unwind." It will compute the sum of the empty list, then the sum of the last list item, then the sum of the last two items, and so on, until it obtains and returns the sum of all items in the list.

In LISP, the body of this function, LISTSUM, would be

```
(COND(EMPTY LIST)0)
  ( T (PLUS(HEADOF LIST)(LISTSUM(RESTOF LIST))))))
```

if we assume appropriate definitions for the functions EMPTY, PLUS, HEADOF, and RESTOF.

Recursion in LISP is the substitute for iteration in more conventional software engineering languages (although a conventional iteration mechanism is also always provided).

One of the most important features of the LISP language for AI purposes is the equivalence between executable code and data. In the example given above, we have a segment of executable code; the body of the function `LISTSUM`. It is also a list of items (a LISP list is delimited by one pair of parentheses).

```
(COND((EMPTY LIST)O)(T(PLUS(HEADOF LIST)(LISTSUM(RESTOF LIST)))))
```

This list is somewhat complex in that it contains sublists, which themselves contain sublists. As a list, it is also a valid LISP data structure in which the list items, for example, the atom `COND` or the sublist `((EMPTY LIST)O)`, can be manipulated by other LISP functions.

Although all lists can be treated as data structures, only certain lists can be executed, namely those lists whose first element identifies a function and the rest of whose elements provide appropriate values for the parameters of the particular function identified by the first element.

As you can see, the syntax of LISP is simple. For both executable code and data, the only structure is a list of objects delimited with parentheses. In fact, the syntax of LISP is too simple; it is so uniform that the inherent structure of a LISP program tends to elude the human perceptual mechanism. What the programmer's eye tells the programmer's brain is that a LISP program is a collection of identifiers that has been liberally sprinkled with left and right parentheses with much the same apparent order as stars in the sky.

But it's great for machines; that is, the syntactic simplicity and uniformity of program and data greatly simplify the implementation of program and data as interchangeable structures.

The demand for dynamic capabilities in AI—run-time modification of both algorithms and data structures—quickly leads to incomprehensible programs. Dynamic systems suggest an interpreter rather than a compiler implementation. Interpreters with their closer proximity to the source code offer a more ready basis for an interactive and helpful programmer support environment than compilers do.

The scenario for AI program development is thus an arrangement of mutual benefit; Desirable language characteristics make the programmer's task more difficult, but the easiest implementation strategy can readily support the addition of modules to ameliorate the programmer's lot.

Extended dialects of LISP are one important manifestation of AI development environments, primarily because of the vast library of support subprograms that constitute an important part of these LISP extensions. The chief reason for developing AI software on a LISP machine such as the `SYMBOLICS 3670` is not the fact that it directly executes LISP but because of the support environment that comes with the machine; a major part of this environment is the 100 MIT man-years of support functions.

Apart from a steady accumulation of support subprograms, there has also been a development of AI-specific control mechanisms and data structures. The development and implementation of new language features gave

gave birth to new AI languages; control structures, such as pattern-directed invocation, in which procedures may be invoked as a result of a pattern-matching operation rather than explicitly by name; and data structures, such as theorems and assertions which, when coupled with the theorem proving control structures, provide a ready implementation for knowledge storage and retrieval systems.

North America has been the spawning ground for most new AI languages, although POP-2 [38] and PROLOG [39] are notable exceptions.

Most AI languages are no longer with us; they exist only as exhibits in the museum of AI. The respective language manuals are material for the historian rather than the serious programmer. Language design in response to needs that were too specific seems to have been the root cause of the lack of longevity in AI languages.

Programming language design, in general, is a poorly understood art: Pascal seems to have been a good one, PL/I a poor one, and ADA remains an interesting open question, although in his Turing lecture, Hoare [40] places ADA in almost the same class as the ill-fated PL/I [41]. In general, Hoare echoes Dijkstra's sentiments that computer scientists should operate only in areas that they clearly understand; this stricture rules out much of AI, almost by definition. With respect to language design, Hoare's major point is that programmers are always surrounded by complexity and "if our basic tool, the language ... is also complicated, the language itself becomes part of the problem rather than part of its solution." Hoare presents a cogent argument for simple, reliable programming languages and for avoiding applications that are not well understood. AI researchers are clearly not fully in accord with Hoare, and we will examine their argument.

An interesting and useful set of principles of programming language design that reflect much of the current wisdom on this topic are given by MacLennan (42).

CONNIVER and PLANNER, for example, were exploratory probes into the space of possible programming languages; they were directed at the area of AI applications. Much useful information was gained from them—information that was vital for mapping the space of AI programming languages, which, of course, are part of the map of all programming languages. Although the map is by no means complete, the original vehicles will not be used again. The information received will facilitate the design, implementation, and launching of the next generation of investigative devices into the AI programming language space. Until much more information is collected and fitted together, the quest for an argot of AI is a fundamentally speculative enterprise.

The search for non-AI programming languages is only slightly more directed because we possess more knowledge of the structure of this space due to relatively prolonged and extensive experimentation within it.

AI programming languages are the most recent descendants from the evolutionary line of man-machine communication. A characteristic trend within this evolutionary sequence is one of abstraction and, thus, a removal of detail that the programmer must be cognizant of (this also implies a loss of fine control, which can be a drawback). From machine code, through assembly codes and high-level languages, to AI languages, we can clearly discern this trend of abstraction.

More conventional computer science has pursued a somewhat parallel course with the development of abstract data structures. Whether the abstraction capabilities implemented in, say, ADA and MODULA-2, will have any direct impact on AI language research remains to be seen.

Before leaving the world of AI programming languages, further mention must be made of PROLOG. It is a European language that provides a neatly defined and efficiently implemented theorem-proving capability. As an efficient inference engine it has a strong claim to be an essential ingredient in the management of knowledge bases for expert systems.

In PROLOG, the control structure and the declarative structure of a program are separated. A PROLOG program specifies only the relationships that hold between the computational objects. How these relationships will be processed (i.e., what control structure will be used) is separate and part of the implementation of the language.

This separation has a number of attractive qualities. We can, for example, "drive" the declarative structure either backward or forward, either deducing the implications of our program or checking whether a given statement is true with respect to the program. But some people would argue that this separating out of control has resulted in a loss of control. Nevertheless, PROLOG represents an interesting new view of programming and one that will undoubtedly be developed in the next few years.

Consider our earlier knowledge base coded as a PROLOG program:

```

NEPHEWOF(Y,X) :- UNCLEOF(X,Y).
HOBBIT(X) :- UNCLEOF(X,Y),HOBBIT(Y).
HOBBIT(bilbo).
UNCLEOF(frodo,bilbo).

```

This PROLOG program is composed of four clauses, which can be read from right to left. Thus, the first clause states that if the UNCLEOF X is Y, then the NEPHEWOF Y is X. The third clause is a fact that states bilbo is a HOBBIT. It should be apparent that this PROLOG program is a straightforward implementation of the knowledge base that we have previously discussed.

If this is a program, then what do the data look like? The user input to a PROLOG program is questions, and the PROLOG program decides the truth or falsity of these questions. Thus, we might ask if the NEPHEWOF bilbo is frodo.

```

?- NEPHEWOF(bilbo,frodo).

```

We are asking the system to check or verify the above statement. Our input statement "unifies" with the conclusion of the first clause when Y is bilbo and X is frodo. In order to conclude the truth of our statement, we must determine the truth of UNCLEOF(frodo,bilbo)—the condition from the first clause obtained by using the same substitution. The truth of this condition is given directly by the fourth clause; thus, the system would output

Yes

In the other mode of using a PROLOG program, we can request the system to compute a result, if possible. We might wish to know what objects are HOBBITs in this system. We would then input

?- HOBBIT(Z).

The system will then attempt to find some value for the variable Z that makes HOBBIT(Z) true. Using clauses two, three, and four, it should then respond

Z=frodo

We can then ask if there are any more known HOBBITs. Using the third clause, it should respond

Z=bilbo

If we ask again, the response should be

No

There appear to be some interesting implications of these two modes—verify and compute—of using a PROLOG program. Analysis [43] has shown that verification of a result is never slower than computing that result, and sometimes it is much faster. The research investigated the possibility of increasing software reliability by using a nonprocedural (e.g., PROLOG) program to check the result of a procedural computation. These analyses may also provide insight into the optimal use of forward- and backward-chaining strategies for knowledge base inferencing in expert systems.

A major reason for the current prominence of PROLOG is the Japanese connection: the Japanese threat to dominate computer technology with "fifth-generation" machines [44], including a proposal that PROLOG might form the basis of the associated programming languages also to be developed. I expect that the outcome will be either a patois based on the best of both AI and non-AI languages or LISP will retain its role as the fundamental language of AI systems.

PROLOG is also prominent in another AI programming language venture: integrated multilanguage systems. POPLOG [45] is such a system and it combines POP-II with LISP and PROLOG. The most important reason for having such a system is that different tasks in a single system can best be served by different languages. So, for example, in the implementation of an intelligent robot, the control algorithms may be best implemented in POP-II, and the knowledge base in PROLOG.

IS INTELLIGENCE NEARLY DECOMPOSABLE?

To return to the possibility of building intelligent systems in limited domains and of developing AI by means of the divide-and-conquer strategy, which is typical of a scientific enterprise, Simon suggested that complex

systems must be nearly decomposable or they will be beyond human comprehension [46]. Near decomposability is a property that enables a complex system to be broken down into (and thus comprehended in terms of) smaller, relatively self-contained subsystems.

How about intelligence: Can we break off pieces and put them in boxes, so that each box contains a high-level, intelligent expert in some very narrow domain? Yes, a number of implemented expert systems constitute an existence proof of that, or do they?

In fact, it is still an open question. What we do know is that even a very limited, low level of general intelligence is exceedingly useful in countless niches of society. In the next few years, we expect to see AI successfully invade these niches. The unknown factor is the longer-term potential for incrementally upgraded specialized expertise. We just don't know the scope and limitations of highly specialized intelligence and the eventual possibilities for domain-specific expertise.

High-level general intelligence might be a critical muddle phenomenon: It might emerge only from a sufficiently large and intermingled knowledge base. Sufficiency in this respect might be orders of magnitude larger and more highly interconnected than the knowledge bases that we are striving to master today. Let's hope not. If the size and complexity of knowledge bases increase exponentially with the level of expertise exhibited by them, we're in trouble.

Figure 1 illustrates some possible relationships between the level of expertise exhibited by expert systems and the size and complexity of the necessary knowledge base. From line *a* to *c*, we go from expert system builder's dream to his/her nightmare. Line *b* represents a function that is perhaps the best realistic hope: Knowledge bases will increase steadily in size and complexity as the level of expertise of the system increases, but not exponentially.

In his exposition of the role of computers in basic education, Papert expounds a reinterpretation of a Piaget theory [47]; the new version is called the "society-of-mind" theory. Fundamental to this theory is the prin-

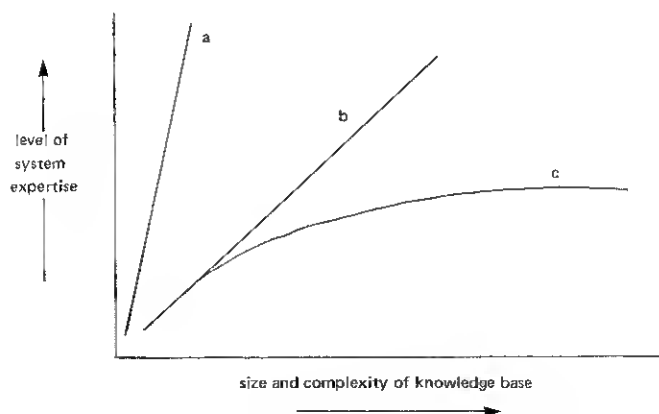


FIGURE 1 Some possible extrapolations of the expert systems phenomenon.

ciple of the epistemological modularity of intelligence; it assumes that intelligence is compounded from the chaos and controversy of independent competing agents.

Perhaps, general intelligence can be readily broken into small chunks ("mind-size bites," Papert calls them), and these chunks will still exhibit a high level of intelligence within more limited domains. We simply must find the right logical hammer. A generalized intelligent system can then be constructed by bolting together a wide range of self-contained, specialized intelligence modules. It won't be that easy, but let's hope that we're at least approximately correct.

Fodor has argued for the existence of "informationally encapsulated" modules in human intelligence [48]. He argues that there is no compelling evidence for nonmodularity despite the many claims to the contrary. A major thesis of the work is that input systems (such as mechanisms of visual perception) are modular in a sense that central cognitive processes (such as belief systems) are not, and furthermore he claims that "the limits of modularity are also likely to be the limits of what we are going to be able to understand about the mind."

From our current perspective the implications are that intelligence may or may not be nearly decomposable or modular. And the extent to which it is representable as a nearly decomposable structure is expected to have an important bearing on whether AI in general will be possible in the form of an understandable and thus maintainable system. Of more immediate consequence is the implication for the possibility of constructing modules that exhibit specialized, intelligent behavior, that is, the future of expert systems, as mentioned earlier.

CAN WE HAVE INTELLIGENCE WITHOUT LEARNING?

The problem of learning can be approached at a number of different levels: from being told a new fact explicitly to deducing some hitherto unrecognized abstract relationship. In terms of the system that does the learning, it may just store a new specific fact or it may integrate into its knowledge a newly discovered abstract concept. The result of learning will be an adaptation of behavior, generally a change for the better.

Given the ill-structured nature of the empirical world and the unforeseeable perturbations that will beset any system that is attempting to behave intelligently, self-adaptation or automatic learning is highly desirable, if not a necessity. Adaptive behavior, from the plant that seeks the light to the parents who raise children, is characteristic of all living organisms. The level and sophistication of adaptive behavior is a useful index of level of intelligence.

Consider the lowly bean randomly falling into the proverbial stony ground. Using this adaptability measure, the average bean seeking up and down for leaves and roots and negotiating awkwardly placed pebbles has far more intelligence than any AI program yet constructed. Clearly adaptive behavior is not the whole story. Equally clearly, it is one fundamental characteristic of intelligence and one that may be critical with respect to the future development of commercial AI.

AI AND LEARNING

Does a useful AI system have to learn? The answer is no. A number of nonadaptive, useful AI systems are extant to buttress this answer. Nevertheless, it is not clear just how far this embryonic movement—the AI invasion of software engineering—can progress and yet eschew the use of self-adaptive mechanisms. And conversely, it is not clear how far the prime goals of software engineering can be met if self-adaptive mechanisms are utilized.

In addition, Michalski et al. argue that machine learning is a practical necessity [49]. It is necessary "to ease the burden of hand programming growing volumes of increasingly complex information," they say.

As stated earlier, there are a number of varieties of machine learning, and they differ in the degree to which they can be readily controlled. I will chart the history of their use in classical AI and point out the implications for commercial AI.

Landmarks and a few particularly illustrative studies are specifically mentioned, but for an exhaustive listing of projects the reader is referred to the bibliographies on machine learning to be found in Ref. 49. In addition, the categories of mechanisms given below are meant to provide some structure in a somewhat amorphous subfield of AI. They are just convenient handles on which to hang the discussion—not mutually exclusive compartments into one of which each project can be correctly stored.

Arthur Samuel's checker-playing program provides an early milestone in machine learning [22]. As we have already described, this program achieved significant practical successes, which were based on its ability to improve its performance as a result of experience.

IF IT MIGHT BE USEFUL, LEARN IT

Rote learning has always been an attractive route to the mechanization of adaptive behaviors. To begin with, it is easy to implement. Just store every new experience, and it works well given the large memories and fast, perfect retrieval typical of computer systems. It also has the important practical advantage that the resultant self-modification can be limited and localized. Hence, it is a learning strategy that offers the best possibility with respect to comprehensibility and, thus, controllability of the overall system.

Unfortunately, the performance of a mechanism based on blind rote learning is soon swamped by the masses of learned data that result. All available storage is eaten up at a prodigious rate, and the system spends most of its time sifting through an ocean of irrelevancies looking for pertinent information: disturbingly reminiscent of the needle-in-the-haystack routine (an all too frequently recurring situation in AI and one that is usually overcome by a few well-engineered, but highly domain-specific, heuristics).

Two possibilities suggest themselves: (a) selective storage and organization for efficient retrieval and (b) storage of generalized information—one general principle rather than many facts.

When he started out in a new building, an expert janitor had to learn where all of the fuse boxes were. He could memorize the facts: In room 100, the fuse box was behind the door; in room 200, it was also behind the door, and so forth. Alternatively, he could learn the general principle that the fuse box is always behind the door of the $x00$ room for each floor x .

There may be a space/time trade-off here between these two learning strategies. If he wants to know where the fuse box is on floor 4, he can, if his learned facts are indexed by floor number, immediately retrieve the relevant fact. But if he uses the latter strategy, actual storage space is less but more time is required to generate the necessary fact from the general principle in conjunction with the particular requirement, fourth floor.

If the janitor opted for storing the general principle about fuse box location, how did he come by it in the first place? For the rote learning scheme, the facts are given and must simply be stored. But the general principle is not given anywhere; it is in some sense implicit in the set of facts.

LEARNING GENERAL PRINCIPLES

There are two major strategies for the machine learning of such general principles: Guess a general principle and then limit it according to subsequent instances of counterexamples, or make the first instance the general principle and generalize it according to the subsequent examples encountered. Both strategies have been used for machine learning in AI and have been discussed recently in Bundy and Silver [50]. This survey is limited to the learning of rules of the form

H_1 and H_2 and ... H_n \longrightarrow Conclusion

that is, a set of hypotheses (the H s) imply a certain conclusion. For example,

(a polygon) and (four sides) and (Christmas Day) \longrightarrow a square

is one such rule, and although not quite perfect, it may be improved by each of the two major strategies, as we shall see.

OVERGENERALIZATION AND REFINEMENT

The discrimination process, as an approach to learning, requires that we first have available a general principle. This first guess may either be generated heuristically from a given instance or fact, or it may be supplied by some outside agent.

With the first approach, our janitor has to come up with a potential general principle about fuse box location when he finds the first fuse box behind the door of room 100. How is he going to do this?

He must generate a sufficiently broad principle—one that can be modified in the light of subsequent failures to find fuse boxes. It must be such that it will converge to a principle enabling him to predict adequately where fuse boxes will be.

So what is his first attempt at this principle? He first finds a fuse box: Fuse boxes are found on walls. Looking around the room at the other walls that do not have fuse boxes, he refines his principle: Fuse boxes are found on one wall of a room. An inspection of all other rooms on this floor leads him to the further refinement: Fuse boxes are found on one wall of one room. Subsequent inspection of other rooms on other floors might finally lead him to the general principle given earlier. That seems easy enough, and a recent study by Langley is an example of machine learning using such a scheme [51].

The crucial information here is distinguishing the important similarities between any two instances of finding a fuse box and distinguishing the important differences between instances of finding and instances of not finding fuse boxes. The similarities and differences are, in fact, endless, but only very few of them are relevant to the process of refining the principle. How do we know which are the important dimensions?

The janitor's first attempt at the rule might have been the following: *Fuse boxes are found on Thursdays.* A few minutes later, he looks into the next room—no fuse box. *Ah, fuse boxes are found on Thursday afternoons at 3 P.M., and so on.* He may eventually exclude some or all of these irrelevant constraints, but will he ever find a useful general principle?

The foregoing smacks of the absurd, but remember we are going to apply these strategies to a computer, not just explain them to a fellow human being. The reason that we find these suggestions unreal is because we know that the location of fuse boxes is invariant with respect to the day of the week and the time of day; Computers don't know that. We have an enormous fund of general knowledge to draw upon. No computer system yet constructed has more than a very small fraction of our general knowledge.

Returning to the earlier rule that aspires to define a square, it is obviously too general: *Not all four-sided polygons are squares even on Christmas Day.* If we present our learning machine with a rectangle on Christmas Day and tell it that this is not a square, we have provided a nonexample that the machine may use to refine its rule.

The crux of this refinement process is to notice that the attributes of the nonexample presented differ from the rule's hypotheses by the fact that its sides are not all equal in length. Hence, we add a further condition to the rule's hypotheses. The new rule is

(a polygon)and(four sides)and(Christmas Day)and(all sides equal)
 —————→ a square

If the machine had computed the essential difference to be a question of angles, the resultant extra hypothesis might have been (all angles not equal). This is not a disastrous mistake, as further examples could be used to eliminate this wrong step (the questions of unlearning mistakes and im-

perfect learning, in general, are both considered later), but, at the same time, it is not much help.

If the machine-learning algorithm had settled on the thickness of the rectangle's sides as the significant attribute, again the resultant rule would be worse rather than better. The machine must be able to select the appropriate attributes of the learning examples presented.

Machine-learning algorithms typically do not cope with the actual examples but only with abstract descriptions of these examples. Winston's famous arch-learning program, for example, was never exposed to actual arches made out of blocks but only to descriptions of arches [52]. And, of course, in a description one can omit the inappropriate attributes.

Here, we see an example of the discontinuity between the problems faced by a system that functions in an abstract domain and one that must work with the real world as it is—abstract AI and concrete AI; they are worlds apart.

Successful machine learning requires, first, that we tell the computer exactly which dimensions of the problem are the important ones, and second, that the difference between the current approximation to the rule and each learning example (or nonexample) is small (the idea of near misses) for teaching purposes).

A FIRST GUESS AND GENERALIZATION

In this, the reverse approach, the system starts with a fairly specific rule and then removes restrictions in the light of subsequent examples.

Using this strategy, the janitor, on finding the first instance of a fuse box, might generate the following tentative rule: *Fuse boxes are found behind the blue door of room 100*. When he later stumbles onto another fuse box behind the green door of room 200, he has found an example that his rule does not cover. Some modification of the rule is called for. The resultant more general rule is the following: *Fuse boxes are found behind the blue door of room 100 and behind the green door of room 200*.

Clearly, this strategy is going to lead to a general rule that is little more than an enumeration of the set of instances encountered. The secret here is to reduce a set of instances to a more concise general rule; we require a mechanism for inductive generalization.

The collection of specific room numbers (100, 200, etc.) should be reduced to something like room $x00$, where $1 \leq x \leq$ number of floors in building. And the set of door colors should be abandoned as nonsignificant. Again we might ask, how did he know that door color was a nonsignificant attribute of his rule? Well, he noticed that there was no regularity in the set of door colors amassed. That might be an answer, but consider how you would specify "regularity" in general terms and algorithmically? Therein lies the problem.

Appropriate generalization of the set of room numbers is not an easier problem, but some inductive generalization algorithms have been successfully applied to certain specific problems. Quinlan surveys a few inductive generalization schemes and describes his own system in some detail [53].

He stresses that the success of current systems is dependent upon describing the instance of the rule by appropriate attributes (i.e., room number rather than time of day for fuse box location). And he goes on to say that the much harder problem of automatically developing good attributes (called "constructive induction") is a problem that will dominate inductive inference research in the 1980s.

Our rule that "defines" a square is also too specific. A further example of a square on New Year's Day, for example, could be used to motivate the necessary generalization, or could it? The modified rule might be

(a polygon)and(four sides)and([Christmas Day]or[New Year's Day])
 —————→ a square

Again, we have much the same problem (except that this time it is finite; there are only 365 different days). As an alternative to the very difficult problem of a widely applicable inductive generalization algorithm, we might use a human tutor to explicitly provide some guidance to the learning machine. There is no requirement that the machine must "go it alone"—humans certainly don't.

Another weakness of some of these general techniques is that they rely critically on both the ordering and the choice of examples. Nevertheless, AI systems that can automatically assimilate knowledge only as a result of the ministrations of a benevolent human tutor could still be of inestimable practical value (and perhaps more readily controlled and understood just because they are externally driven).

On the other hand, external tutoring puts a considerable burden on the human tutor. In fact, we are beginning to see a revival of interest in more sophisticated machine learning initiated by expert systems' builders, just because this burden is becoming intolerable.

Bundy and Silver warn us that neither of these two general strategies is complete in itself (i.e., we cannot know if we have arrived at the optimal rule); we need to apply both strategies (they suggest generalization before discrimination) to be sure of arriving at the best rule [50].

LEARNING BY BEING TOLD

Given that AI programs tend to be both large and lacking in simple general principles, machine learning must be treated with circumspection. If learning is initiated and controlled by user interaction, we have the best chance of maintaining control of the resultant beast. This type of scheme can also be the easiest to implement, because much of the burden remains with the human tutor.

As we shall see later, the development of an AI program is necessarily an incremental process. Learning by being told, or "advice" taking as it is sometimes called, is a potentially very useful way of automating some aspects of this incremental development. It can thus play a lead role in system development rather than just being an inessential member of the supporting cast.

Of particular interest in this respect is the process of expert system development. An incremental process of feedback and evaluation is used to refine and extend expert system knowledge. Two essential parties are the human expert, with advice to give, and his mechanized counterpart, with advice to be woven into its current knowledge base. It is, of course, highly unlikely that these two potential communicants share a common language. So a computer expert or "knowledge engineer" is also required as a go-between; his/her humanness ensures a degree of communication with the human expert, and the actual system is his/her baby, which ensures that he/she can communicate with it, if anybody can.

The TEIRESIAS project [7] is an attempt to shorten this chain of command and thereby allow direct human expert-computerized expert conferences (which, incidentally, releases the computer science expert, a scarce resource). Davis' and Lenat's system interfaces between a human physician and the expert medical system MYCIN.

One of the interesting aspects of TEIRESIAS is that it seizes command of the dialogue whenever possible; this all too human trait has several advantages even for a computer. In particular, it both simplifies the natural language interface by leading the human responses into situations where there are few possible responses and tries to ensure that advice is given at the level and in the sort of chunks that it can digest most easily.

Once again, we must wait for time, the great leveler, to let us know if TEIRESIAS really heralds a wave of knowledge engineers out of work or whether the time and effort needed to tune a TEIRESIAS for each particular expert system will just increase their scarcity value. In principle, it is true that a generalized TEIRESIAS need only be constructed once, but such generalized solutions to AI problems have eluded the researchers in the field so far.

LEARNING BY INTROSPECTION

This category of machine learning refers to the possibility of generating new ideas from old. That is, while driving home after a busy night taking care of his office block, the janitor might ruminate upon his current stock of job-related knowledge—a sort of mental chewing of the cud, masticating observations, plans both successful and unsuccessful, stored facts, and so forth.

What is the upshot of all these mental gymnastics, you might well ask? The diverse products will be general rules, new concepts, new plans, and improved problem-solving strategies—a bountiful harvest.

So let's get our AI to do some of this in its slack periods, if not in parallel while it is working flat out. And indeed, there are a number of lines of AI research that have investigated and are still investigating this brand of machine learning.

Over 10 years ago (54), this technique of machine learning was successfully employed within an adaptive FORTRAN translator. The translator contained an AI module that periodically surveyed the results of the prior parsing session and then reorganized its knowledge base. One of the tasks

of this module was to remove unwanted generalization by finding the most frequently used particular instances of a general rule and promoting them to positions of higher precedence than the general rule itself.

The system learned to expect the specific idiosyncrasies of a given operating environment. The resulting behavioral changes were both an increase in general efficiency and an increase in quality of error correction.

Working with introspection in the opposite direction, that is, from particular instances to a general rule, is more difficult. Inductive generalization falls into this category.

A particularly interesting application of this technique is described by Michie [55]. He considers the possibility of constructing an expert system backwards because it is easier that way. An expert system can be thought of as a function from the input problem space to the expert solution space. This forward transformation is typically difficult, but sometimes the reverse transformation, the inverse function, from a given solution to the problem that would have generated it, is easily computable.

So if we can generate the space of solutions, we do so and store each one away with its easily computed problem. This bunch of solution-problem pairs is then used in the forward direction. It is an expert system. However, as you may have guessed, there is a snag. Our expert system is going to be impractically large. So Michie suggests that we loose an inductive generalization algorithm on our masses of problem-solution pairs and thereby reduce this unwieldy conglomeration to a manageable number of general rules.

An inductive algorithm identifies and extracts the significant commonalities exhibited by a set of instances (and perhaps lacking in a set of non-instances). The set of instances is then replaced by the resulting general principle.

A nontrivial learning device, human or machine, must have a complex goal structure, both immediate goals and long-term goals. It is this set of goals in conjunction with stored knowledge that singles out a very few aspects of sensory experience as significant events—events that should be given special attention in order to improve performance as a result of experience.

In the development of an expert system for soybean disease diagnosis, Michalski and Chilausky inductively derived the diagnostic rules from a collection of instances of symptoms and the diagnosed disease [56].

To the delight of most observers (except expert soybean disease taxonomists), the system using the inductively generated rules was more accurate than a system using the official rules of soybean disease taxonomy. This leads us a little aside, but in AI, field trials (excuse the pun) are somewhat scarce events so we should make the most of the ones we have.

There are two points here: First, Michie's scheme is shown to be practically viable, and second, human expertise should be extracted in the form of specific examples, not general rules.

UNLEARNING MISTAKES

Nontrivial learning in an ill-structured domain, such as the empirical world, cannot always be correct. The learning system will need the ability to un-

learn the things that were incorrectly learned. This is an awkward problem for a number of reasons.

1. The fact (more likely, suspicion) that something erroneous has been learned need not be apparent until long after the actual learning event.
2. Even if we are sure that some learned behavior is incorrect, correct identification of the culprit component(s) in a complex structure can be a very difficult problem—in effect, the credit assignment problem once more.
3. Having identified the incorrectly learned feature, we must remove its adverse effects without destroying the complete structure in which it might be embedded.

Hayes-Roth et al. suggest and give examples of the use of metaknowledge (knowledge about the domain-specific knowledge of their expert systems) for identifying incorrect rules learned from an expert [57]. The system's metaknowledge brings suspicious rules (such as ones in which the antecedent condition can never be satisfied) to the attention of the knowledge engineer. If the knowledge base is truly a modular collection of such rules, then removal or modification of the erroneous piece of knowledge is a relatively simple process.

This example raises two points that we should note.

1. Modularity is a key aspect of manageable AI software.
2. Metaknowledge is essential for the effective control of nontrivial machine learning.

Langley simplifies the three problems of "unlearning" listed above by setting up his system so that the correctness of a proposed addition will be known immediately and can thus be removed if it is incorrect [51]. But, in general, we cannot expect situations with the luxury of immediate assessment.

It is instructive to examine what we know of the human solution to this unlearning problem. A first point to note is that humans are not particularly good at one-trial learning—we generally require repetition before we learn something. This is a difference between humans and machines that is often touted as part of an argument that computers can be expected to eventually surpass humans in intelligent abilities. But from our current viewpoint, there may be an advantage in resistance to one-trial learning: If before we learn it, we require a certain number of repetitions of an event, there is a chance to assess its correctness before a full commitment to learn it is made.

Studies of skill acquisition definitely suggest a two-stage learning process at least. In the first stage, we tentatively learn a new skill, and only when this tentative learning is confirmed as correct by repeated use is the skill knowledge "compiled" into a more efficient, but less modifiable, form. Anderson proposes such a model of skill learning [58]: "a declarative stage in which facts about the skill domain are interpreted and a procedural stage in which domain knowledge is directly embodied in procedures for performing the skill."

The analogy with the trade-offs between interpreting and compiling computer programs is obvious and instructive.

To finish the section on machine learning, still properly under the rubric of learning by introspection, let us briefly look at an exciting development, even though it is probably the least practical (in terms of immediate application in practical software).

MECHANIZED CREATIVITY

AI, as is appropriate for preliminary studies, has so far concentrated on mechanized discovery within the much more restricted and well-structured domains of mathematics and classical chemistry.

Human versus machine creativity is a fascinating and usually provocative subject, typically engendering copious amounts of heat and sometimes a little light. So rather than venture into this supercalorific domain, let us do little more than just observe that active investigation of the possibilities for mechanized discovery of new and unanticipated principles (as opposed to the relatively expected and foreseen results of inductive generalization) is well under way in AI (see the AM system in Ref. 7).

This is important because of the general importance of machine learning, and it is perhaps dangerous because of its very nature. The results are, almost by definition, unexpected. This does not mean that they might not be extremely useful. What it might mean is that we must be apprised of each discovery and investigate its scope and limitations before we use it within an application program.

So much for machine learning. Clearly, there is a lot of scope for practical potential. There are definite areas of light and even a very few and limited practical applications. But in self-modifying systems, as every software engineer knows, there is also a lot of scope for disaster.

THE METHODOLOGY OF AI SYSTEM DEVELOPMENT

AI problems tend to be dynamic and poorly structured; they also tend to be characterized in terms of a behavioral description, that is, rather than having an abstract definition of the semantics of English, we have a description of how English-speaking people are likely to respond to various classes of utterance. An adequate English-understanding AI system will respond in similar ways. Thus, in AI system development, we are not so much implementing some well-specified problem as exploring the adequacy of a machine-executable specification (i.e., a program) vis-a-vis a dynamic and ill-structured, performance-mode problem description.

AI systems, particularly those of mainstream and of scientific AI, tend to be developed in an incremental and exploratory manner. The methodology for AI system development is based on the run-debug-edit cycle: A program is executed, its behavioral inadequacies are analyzed, and the program is redesigned to eliminate the perceived inadequacies; the modified program is executed, and so forth. Currently, there is much debate as to how to improve this incremental development paradigm or even whether the

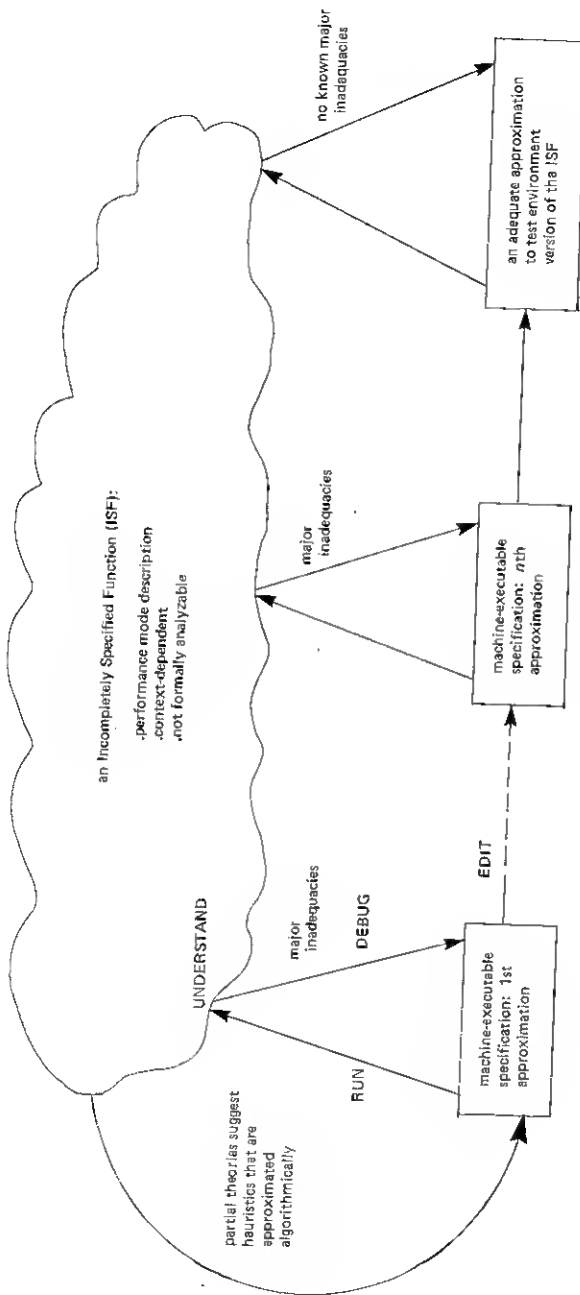


FIGURE 2 The AI program development process.

incremental phase can be prior to any implementation, if not eliminated altogether (see Refs. 59, 60, and 61). Figure 2 is a schematic illustration of an incremental system development methodology in AI.

THE FUTURE OF AI

Finally, I have emphasized the difficulties and unresolved problems that exist in the field of AI; there are many of them and they appear to be difficult. Nevertheless, certain aspects of AI (e.g., limited knowledge bases) have been explored with considerable success in recent years. The market for the commercialization of certain AI techniques is both rich and broad. But there do appear to be severe inherent limitations in the current expert system's technology that will block advances both to higher levels of expertise and to applications beyond abstract technical expertise, the domain of present success. Dreyfus and Dreyfus, for example, argue for such inherent limitations [62]. It should also be realized that before we will see anything like the full potential of AI in practical software (as opposed to research projects), a number of nontrivial technical problems need to be solved—problems such as robust and reliable, nontrivial machine learning—and more generally we need a discipline of incremental system development that will yield robust, reliable, and readily comprehensible software products. At present, we have little idea of how to achieve such goals.

REFERENCES

1. J. McDermott, "R1: The Formative Years," *AI Mag.* 2(2), 21-29 (1981).
2. A. Bundy, "The Nature of AI: A Reply to Ohlsson," *AISB Q.*, (47) 24-25 (1983).
3. A. Bundy, "The Nature of AI: A Reply to Schank," *AI Mag.* 4(4), 29-31 (1983).
4. C. Hewitt, "The Challenge of Open Systems," *Byte*, 223-242 (April 1985).
5. A. M. Turing, "Computing Machinery and Intelligence," *Mind* (59) 433-460 (1950).
6. J. Weizenbaum, "ELIZA—A Computer Program for the Study of Natural Language Communication Between Man and Machine," *Comm. ACM* 9 (1), 36-45 (1965).
7. R. Davis and D. B. Lenat, *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, New York, 1982.
8. J. Weizenbaum, *Computer Power and Human Reason*, Freeman, San Francisco, CA., 1976.
9. D. Partridge, "The Social Implications of AI," in *Artificial Intelligence: Applications and Implications* (M. Yazdani, ed.), Chapman and Hall, London, England, 1986, pp. 315-336.
10. H. A. Simon, "Prometheus or Pandora: The Influence of Automation on Society," *IEEE Comp.* 69-74 (November 1981).
11. E. B. James and D. Partridge, "Machine Intelligence: The Best of Both Worlds?" *Int. J. Man-Mach. Stud.*, 4, 23-31 (1973).

12. D. R. Hofstadter, Godel, Escher, Bach: An Eternal Golden Braid, Basic Books, New York, 1979.
13. A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, Vol. 2, Kaufman, Palo Alto, CA, 1982.
14. M. A. Arbib, *The Metaphorical Brain*, Wiley, New York, 1972.
15. V. S. Johnston, D. Partridge, and P. D. Lopez, "A Neural Theory of Cognitive Development," *J. Theoret. Biol.*, 100, 485-509 (1983).
16. D. Partridge, "Input-Expectation Discrepancy Reduction: A Ubiquitous Mechanism," *Proc. 9th International Joint Conference on AI*, UCLA, 1985, pp. 267-273.
17. D. Partridge, J. E. MacDonald, V. S. Johnston, and K. R. Paap, *AI Programs and Cognitive Models*, Tech. Report MCCS-86-60, New Mexico State University, Las Cruces, NM, 1986.
18. D. Michie, "The State of the Art in Machine Learning," in *Introductory Readings in Expert Systems* (D. Michie, ed.), Gordon and Breach, London, 1982, pp. 208-229.
19. J. Pearl, *Heuristics*, Addison-Wesley: Reading MA, 1984.
20. W. N. Locke and A. D. Booth, eds., *Machine Translation of Languages*, MIT and Wiley, New York, 1955.
21. B. Raphael, *The Thinking Computer*, Freeman, San Francisco, CA, 1976.
22. A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," in *Computers and Thought* (E. A. Feigenbaum and J. Feldman, eds.), McGraw-Hill, New York, 1963, pp. 71-105.
23. B. Buchanan, G. Sutherland, and E. A. Feigenbaum, "Heuristic Dendral: A Program for Generating Explanatory Hypotheses in Organic Chemistry," in *Machine Intelligence 5*, (B. Meltzer and D. Michie, eds.), Elsevier, New York, 1969.
24. E. H. Shortliffe, *Computer-based Medical Consultation: MYCIN*, Elsevier, New York, 1976.
25. M. Minsky, "A Framework for Representing Knowledge," in *Mind Design*, (J. Haugland, ed.), MIT Press, Cambridge, MA, 1981, pp. 95-128.
26. J. Mylopoulos and M. J. Levesque, "An Overview of Knowledge Representation," in *On Conceptual Modelling*, (M. L. Brodie, et al., eds.), Springer-Verlag, New York, 1984, pp. 3-17.
27. W. A. Woods, "What's in a Link: Foundations for Semantic Networks," in *Representation and Understanding* (D. G. Bobrow and A. Collins, eds.), Academic Press, New York, 1975, pp. 35-82.
28. D. Partridge, "A Syntactic View of Semantic Networks," *Int. J. Man-Mach. Stud.*, 10, 113-119 (1978).
29. P. J. Hayes, "The Naive Physics Manifesto," in *Expert Systems in the Electronic Age* (D. Michie, ed.), Edinburgh University Press, Edinburgh, 1979, pp. 242-270.
30. B. Chandrasekaran and S. Mittal, "Deep Versus Compiled Knowledge Approaches to Diagnostic Problem-Solving," *Int. J. Man-Mach. Stud.*, 19, 425-436 (1983).
31. H. J. Levesque, "The Logic of Incomplete Knowledge Bases," in *On Conceptual Modelling* (M. L. Brodie, et al., eds.), Springer-Verlag, New York, 1984, pp. 165-186.

32. A. Collins, E. H. Warnock, N. Aiello, and M. L. Miller, *Reasoning from Incomplete Knowledge in Representation and Understanding* (D. G. Bobrow and A. Collins, eds.), Academic, New York, 1975, pp. 383-415.
33. J. R. Carbonell, "AI in CAI: An Artificial Intelligence Approach to Computer-Aided Instruction," *IEEE Trans. Man-Mach. Syst.*, MMS-11, 190-202 (1970).
34. J. McCarthy et al., *LISP 1-5 Programmer's Manual*, 2nd ed., MIT Press, Cambridge, MA, 1965.
35. D. S. Touretzky, *A Summary of MacLisp Functions and Flags*, 4th ed., Carnegie-Mellon University Computer Science Department, Pittsburgh, PA, 1982.
36. W. Teitelman and L. Masinter, "The INTERLISP programming environment," *IEEE Trans. Comp. C-14*(4), 25-35 (1981).
37. E. Sandewall, "Programming on an Interactive Environment: The 'LISP' Experience," *Comput. Surv.*, 10(1), 35-71 (1978).
38. R. J. Popplestone, "The Design Philosophy of POP-2," in *Machine Intelligence 3* (D. Michie, ed.), Edinburgh University Press, Edinburgh, 1967.
39. W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*, Springer-Verlag, New York, 1981.
40. C. A. R. Hoare, "The Emperor's Old Clothes," *Commun. ACM*, 24(2), 75-83 (1981).
41. E. W. Dijkstra, "The Humble Programmer," *Commun. ACM*, 15(10), 271-280 (1972).
42. B. J. MacLennan, *Principles of Programming Languages*, Holt, Rinehart, and Winston, New York, 1983.
43. L. Guthrie, "Some Results on Evaluating and Checking Functions in a Language Designed for Redundancy," Ph.D. thesis, New Mexico State University, Las Cruces, NM, 1985.
44. E. A. Feigenbaum and P. McCorduck, *The Fifth Generation*, Addison-Wesley, Reading, MA, 1983.
45. A. Sloman and S. Hardy, "POPLOG: A Multi-Purpose Multi-Language Program Development Environment," *AISB Q.*, (47), 26-34 (1983).
46. H. A. Simon, "The Architecture of Complexity," *Proc. Amer. Philosoph. Soc.* 106(6), 467-482 (December 1962).
47. S. Papert, *Mindstorms*, Basic Books, New York, 1980.
48. J. A. Fodor, *The Modularity of Mind*, MIT Press, Cambridge, MA, 1983.
49. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine Learning*, Tioga, Palo Alto, CA, 1983; Vol. II, 1986.
50. A. Bundy and B. Silver, "A Critical Survey of Rule Learning Programs," *Proc. ECAI*, 151-157 (1982).
51. P. Langley, "Learning Search Strategies through Discrimination," *Int. J. Man-Mach. Stud.*, 19, 512-541 (1983).
52. P. H. Winston, "Learning Structural Descriptions from Examples," in *The Psychology of Computer Vision*, (P. H. Winston, ed.), McGraw-Hill, New York, 1975.
53. J. R. Quinlan, "Semiautonomous Acquisition of Pattern-Based Knowledge," in *Introductory Readings in Expert Systems* (D. Michie, ed.), Gordon and Breach, London, pp. 192-207, 1982.

54. D. Partridge, "A Dynamic Database which Automatically Removes Unwanted Generalization for the Efficient Analysis of Language Features That Exhibit a Disparate Frequency Distribution," *Comp. J.* 18(1), 43-48 (1975).
55. D. Michie, Transcription of a Lecture, UCLA (March 9, 1983).
56. R. S. Michalski and R. L. Chilausky, "Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *Policy Anal. Inf. Syst.*, 4(2), 125-160 (June 1980).
57. F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, *Building Expert Systems*, Addison-Wesley, Reading, MA, 1983.
58. J. R. Anderson, unpublished information, 1982.
59. R. Kowalski, "AI and Software Engineering," *Datamation* 30(18) (1984).
60. D. Partridge, "Engineering AI Software," *AI Rev.*, 1(1), 27-41 (1986).
61. J. Mostow, "Response to Derek Partridge," *AI Mag.* 6(3) pp. 51-55 (1985).
62. H. L. Dreyfus and S. E. Dreyfus, *Mind Over Machine*, Free Press/Macmillan, New York, 1986.

BIBLIOGRAPHY

Boden, M. A., *Artificial Intelligence and Natural Man*, Basic Books, New York, 1977.

A refreshingly new look from an (at the time) outside observer, based on detailed studies of a number of important programs.

Bundy, A., ed., *Artificial Intelligence Tools*, Springer-Verlag, New York, 1984.

A catalog of AI programming tools and techniques. Contains a short description in English of each tool and a reference to where further details may be found (see Charniak et al. [1980] for the LISP view).

Bundy, A. and B. Silver, "A Critical Survey of Rule Learning Programs," *Proc. ECAI*, 151-157 (1982).

Contains a detailed analysis and comparison of generalization and discrimination machine-learning strategies.

Campbell, J. A., ed., *Implementations of PROLOG*, Ellis Horwood, United Kingdom, 1984.

A collection that focuses on PROLOG, the language of logic programming and the language that will form the basis of the Japanese "fifth generation" project.

Charniak, E. and D. McDermott, *Introduction to Artificial Intelligence*, Addison-Wesley, Reading, MA, 1985.

Probably the next standard introduction to the field. It uses knowledge representation in the predicate calculus as the basis for a unified theory.

Charniak, E., C. K. Riesbeck, and D. V. McDermott, *Artificial Intelligence Programming*, Erlbaum, NJ, 1980.

An attempt to collect the accepted core of practical programming techniques in AI. The techniques are presented in detail in terms of the

LISP language (see Bundy [1984] for a more general, but only descriptive, treatment of AI techniques).

Davis, R., and D. B. Lenat, *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, New York, 1982.

It should have been entitled "Two Knowledge-Based Systems..."

It is divided between TEIRESIAS and AM, but there is a lot of information on the two systems.

Dreyfus, H., *What Computers Can't Do*, Harper and Row, New York, 1972; 2nd ed., 1979.

Fundamentally wrong, I think; nevertheless, it is the standard work and contains a lot of good information but also a lot of flaws. Should be tempered with Z. W. Pylyshyn, *Cognition* 3(1), 57-77 (1975).

Argued much better in the second edition.

Feigenbaum, E. A., and A. Barr, *The Handbook of Artificial Intelligence*, Kaufmann, CA, 1981-1982, Vols. I, II, and III (edited by P. R. Cohen, in place of Barr).

A bid for the standard reference work in AI, and it does a good job except that it spares space for neither the biological and neurophysiological beginnings nor the ongoing research that falls outside the heuristic programming approach.

Feigenbaum, E. A., and J. Feldman, eds., *Computers and Thought*, McGraw-Hill, New York, 1963.

A classic early collection of papers. It contains both Turing's classic paper and Samuel's study in machine learning, as well as many others.

Hayes-Roth, F., D. A. Waterman, and D. B. Lenat, *Building Expert Systems*, Addison-Wesley, Reading, MA, 1983.

A useful attempt to abstract from the plague of particular expert systems in order to convey general structure and principles and describe software that is aimed at facilitating the construction of future computerized experts.

Hofstadter, D. R., *Godel, Escher, Bach: An Eternal Golden Braid*, Basic Books, New York, 1979.

The best AI book ever! A multilevel extravaganza on minds and machines (and many other things). Thought provoking and droll investigations of all aspects of AI.

Jackson, P. C., *Introduction to Artificial Intelligence*, Petrocelli, New York, 1974.

The first (and last) attempt to cover all approaches to AI; it has a pleasing breadth but a (perhaps inevitable) lack of depth.

McCarthy, J., *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, MA, 1962.

The basic reference to the LISP language written by the language designer but probably not the easiest book from which to learn LISP. McCorduck, P., *Machines Who Think*, Freeman, San Francisco, CA, 1979.

The secret history of artificial intelligence. A fascinating study mostly straight from the scientists' mouths. The reader might detect a certain bias toward the home territory, but that is to be expected. McDermott, D., "Artificial Intelligence Meets Natural Stupidity," *SIGART Newsletter*, (57), 4-9 (April 1976).

A severe criticism of AI oversell and the Ph.D.-driven research machine that accounts for most of it.

Michalski, R. S., J. G. Carbonell, and T. M. Mitchell, *Machine Learning*, Kaufman, Los Altos, CA, Vol. I, 1983; Vol. II, 1986.

Up-to-date description of a number of significant projects. Contains a large bibliography on machine learning.

Michie, D., ed., *Introductory Readings in Expert Systems*, Gordon and Breach, London, 1982.

More of a Euro-view of expert systems. A small but broad collection of articles aimed at introducing the technical nonspecialist to this field.

Michie, D., and R. Johnston, *The Creative Computer*, Penguin Books, United Kingdom, 1984.

A very optimistic view of the societal and technological benefits that machine intelligence will bring in the near future.

Narayanan, A., and N. E. Sharkey, *An Introduction to LISP*, Ellis Horwood, United Kingdom, 1985.

One of the many books on the fundamental programming language of AI, that is, LISP.

Nilsson, N. J., *Learning Machines*, McGraw-Hill, New York, 1965.

A detailed mathematical treatment of machines that can be trained to classify patterns.

Nilsson, N. J., *The Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

A detailed exposition of the technical principles of searching and structuring that have been developed in AI, which can be viewed as the major principle, but not much more than his earlier book, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.

Partridge, D., *Artificial Intelligence: Applications in the Future of Software Engineering*, Ellis Horwood and Wiley, United Kingdom, 1986.

An attempt to expose the current problems, as well as the potential benefits, of AI in commercial software. The encyclopedia entry is derived from several chapters of this book.

Raphael, B., *The Thinking Computer*, Freeman, San Francisco, CA, 1976.

An exposition of AI's major subfields, which attempts to tread the middle ground between superficiality and technical detail.

Ringle, M. D., ed., *Philosophical Perspectives in Artificial Intelligence*, Humanities Press, NJ, 1979.

A collection of viewpoints, from both philosophers and AI researchers—some supportive and some critical of AI.

Samuel, A., "Some Studies in Machine Learning Using the Game of Checkers," in *Computers and Thought* (E. A. Feigenbaum and J. Feldman, eds.), McGraw-Hill, New York, 1963, pp. 71-105.

Both rote and parameter learning; a great study years ahead of its time. A study containing detailed empirical results: a paradigm that has unfortunately not endured. The anthology, *Computers and Thought*, is itself rich ground for the AI archaeologist.

Simon, H. A., *The Sciences of the Artificial*, MIT Press, Cambridge, MA, 1969.

A small but stimulating collection of lectures given by a pillar of the AI establishment.

Sparck-Jones, K., and Y. Wilks, eds., *Automatic Natural Language Parsing*, Ellis Horwood, United Kingdom, 1985.

A summary of the current wisdom on this first part of the very difficult problem of computers understanding natural language.

Weizenbaum, J., *Computer Power and Human Reason*, Freeman, San Francisco, CA, 1976.

A social argument on the lines of what AI researchers should not do. Not as clear cut a question as Weizenbaum tries to make it; nevertheless, ethical and moral questions in AI are important today and are likely to become more so in the future.

Winston, P. H., *Artificial Intelligence*, Addison-Wesley, Reading, MA, 1977; 2nd ed., 1984.

An introductory AI textbook. Covers the famous AI programs and their mechanisms in some detail.

DEREK PARTRIDGE

ARTIFICIAL INTELLIGENCE FOR AUTOMOBILE MANUFACTURING

INTRODUCTION

Until recently, the field of artificial intelligence (AI) had remained largely within the esoteric confines of major research universities where it was nurtured for over two decades. Though a few successful expert systems were developed even more than a decade ago, primarily in academia, large-scale commercialization in industry began only within the last 3 to 5 years. Today, AI technology is moving from the research laboratories into the commercial domain with great momentum. The excitement from the commercialization of AI technology permeates user, vendor, and research communities alike. The user community in the automobile industry is no exception. Facing fierce competition, most members of the giant automobile industry are embracing AI as yet another potentially powerful means to gain a competitive advantage.

Often the term artificial intelligence is used loosely to mean knowledge engineering or expert systems. In fact, AI represents a collection of technology areas, including knowledge engineering, robotics, machine vision, natural languages, and voice recognition, among others. Robotics and machine vision applications have been in existence on the plant floors of the automobile industry for quite some time. On the other hand, in terms of commercial emergence, knowledge engineering is the most recent element of AI to move from the research laboratories to commercial implementation. It is among the fastest growing subfields of AI and is being viewed by the automobile industry as having great potential for enhancing the overall productivity in the manufacturing business.

This article focuses on the knowledge engineering component of AI. Features of knowledge engineering are described and contrasted with conventional programming techniques. The specialized computer hardware needed for developing and hosting expert systems, the tools or shells used for building these systems, and the computer languages underlying the tools are discussed. A brief description of the internals of AI technology with reference to the automobile industry is included. Special skills needed for knowledge engineers in the automobile industry and desired educational backgrounds are outlined. The need for and the effects of strategic relationships between automakers and specialized AI companies are also described. As the automobile industry begins to field expert systems in its plants, issues pertaining to reliability, validation, and acceptance of those systems will become very crucial. These factors are covered briefly in this article. Finally, the shortcomings of the technology, the research under way, and the promise of the future in the automobile industry are discussed.

Virtually every area of the automotive business has potential applications where AI technology could be utilized to advantage. These application



FIGURE 1 Application in automobile servicing and repair.

areas range from general troubleshooting to intelligent designing and planning to sales and service. Figure 1 shows a sample service application presently being pursued by automakers both in the United States and abroad. It involves using AI to assist auto technicians in diagnosing and repairing automobile engine problems effectively. Potential areas for AI application include diagnosing of electronic, electrical, and mechanical components of automobiles and of the machines that manufacture them. Figure 2 shows an example of a potential application of AI in monitoring manufacturing equipment and performing preventive maintenance. Monitoring and control of discrete and continuous processes, hydraulic and pneumatic systems, lubrication and cooling systems, and sensor data verification are examples of some other potential manufacturing-related applications. Applications to machine controllers, robotics, machine vision, scheduling, and simulation and selection, to name a few, are also being considered.

Some of the areas of automobile manufacturing are highly technologically advanced, and the rapid changes in them often leave the workers trailing behind. At the other extreme, many of the tasks in the manufacturing en-



FIGURE 2 Application in manufacturing equipment and maintenance.

vironment are more of an art than a science. These are generally handled on the factory floors only by a few select artisans. In addition, the workforce demographics in the auto industry is adversely affected due to retirements and normal attrition. AI systems can help alleviate these problems by capturing, retaining, and quickly disseminating pertinent knowledge.

KNOWLEDGE ENGINEERING

Knowledge engineering is the process of capturing human knowledge and expertise and coding it in the form of computer programs. The program so obtained is generally referred to as a knowledge system, or more commonly, an expert system.

Conventional programs and expert systems differ in many ways. Some differences are apparent at a glance, whereas others, mainly the key ones, are more subtle. The principal difference between the two is in the nature or style of the programs. Expert systems in the auto industry and else-

where are traditionally written in a symbol-manipulating language, such as LISP or PROLOG, or higher-level tools built on these languages. They deal primarily with symbolic data as opposed to the numeric data used in conventional programs, such as those written in BASIC, FORTRAN, Pascal, and so forth. This feature facilitates AI programs in exhibiting "intelligent and humanlike behavior." Data (e.g., the condition of a car or machinery) plays a crucial role in the execution of expert system programs. Different data inputs can provide a totally different path of execution of an expert system program. In conventional systems, data play only a minor and, at best, a secondary role compared with the program.

AI programs in the auto industry mimic the flexibility of reasoning done by engineers, manufacturing specialists, and shop floor personnel, resulting in robust systems that are capable of handling ill-defined problems. In the manufacturing environment, such systems handle incomplete and, at times, even inconsistent data. In addition, these systems can and usually do operate on uncertain data. More often than not, in the manufacturing environment, a complete picture of the problem is not known in advance. It emerges only as the AI program is incrementally developed and expanded. This feature of modularity and incremental development enables rapid prototyping and testing of expert systems.

Expert systems are programmed so that they can draw conclusions that are not explicitly coded. Though present expert systems in the automobile industry are rather weak in this area, they can, to some extent, generalize to form new concepts and can thus recognize new, complex patterns.

Most of the expert systems in use in the industry are made up of heuristics or "rules of thumb" as opposed to precise algorithms. The knowledge and the inference mechanism that operates on that knowledge are almost always separated from one another. Thus, both knowledge and control can be individually altered without affecting the other.

As an additional benefit, expert systems are being considered for training and learning purposes in the automobile industry. Finally, because of the nature of the problems handled (large complex problems requiring simulation of human judgement), a typical expert system usually yields a larger monetary return in the automobile manufacturing business than a similarly sized conventional system does.

COMPUTER HARDWARE

A full spectrum of computer hardware has been used for AI applications in the automobile industry. Expert systems have been developed on conventional computers, ranging from large mainframes, minicomputers, and engineering workstations to personal computers. Each has its place in helping to proliferate the use of AI technology in the automobile industry. The type of hardware used depends on the size, cost, and the complexity of the system to be developed. Small systems can be developed on a desktop personal computer, whereas medium- to large-size systems can be built on engineering workstations. For large, complex systems, specialized LISP machines continue to be the first choice for program development.

AI applications in automobile manufacturing can be extremely large and complex. Developing expert systems for such applications often necessitates dedicated LISP machines with large RAM and disk capacities, high execution speeds, and powerful development environments. Upwards of 8Mb of main memory and over 200Mb of disk storage is not uncommon for developing large expert systems in the automobile manufacturing area.

For many automobile applications, availability of a good programming environment is a necessity because the domain expert and the knowledge engineer are usually two different people. The system is developed on an incremental basis, and within each increment, the knowledge engineer goes through several alterations of the program. Under such a situation, programmer productivity becomes a critical factor. Fortunately, today's LISP machines provide some of the best programming environments for the user's convenience. These include features such as high-resolution graphics, multiple and arbitrary-size windowing capabilities, large physical screens, mouse and menu systems, powerful full-screen editors, dynamic linking and loading of programs, debugging and tracing facilities, and efficient automatic memory management.

Tightly integrated facilities for languages such as BASIC, FORTRAN, Pascal, and even PROLOG are available on dedicated LISP computers. These computers, though tailored primarily for symbolic computation, perform numeric operations with effectiveness similar to those obtained with conventional computers. This is especially important in automotive applications where several interspersed engineering calculations may have to be performed during the course of an expert system consultation. If necessary, floating-point accelerators can be added to speed up the arithmetic. Similarly, to speed up the execution of the LISP code, direct microcode compilation is made available to the user. Though sophisticated architectures using 36-bit words have been in use, most of these machines use 32-bit words. Some of these machines have additional 32-bit non-LISP coprocessors that support the UNIX operating system. A choice of several standard conventional languages is made available on the coprocessor.

Though AI machines are generally used in a single-user, personal computer format, there are good facilities for networking several of them together. They can communicate and network with other conventional computers as well. Standard RS-232 links providing networking capabilities with IBM mainframes through systems network architecture (SNA) are now available. Similarly, standard ethernet links, providing access to a wide range of DEC machines via DECNET and TCP/IP protocols, are available. Communications over the manufacturing automation protocol (MAP) will become important because a large amount of factory control and computer communications in the automotive business are planned to be executed via MAP in the future. These networking/communications facilities are crucial for effective application of AI systems in the automobile industry.

The need for fast execution speeds is a function of the nature of the application. Highly sophisticated, state-of-the-art computer-controlled machinery performing high-speed, precision jobs is commonplace on the automobile factory floors today. Intelligent, real-time monitoring, diagnosing, maintaining, or controlling such machine equipment will require a significant increase in the speeds of inferences of current expert systems.

Alternatively, in the future, such machine equipment may have to be redesigned to include the AI systems in an integral fashion. LISP chips are expected to be commercially available and could be integrated with the conventional computers presently running the machinery. Yet another option is to translate the expert systems requiring high execution speeds to equivalent hardware chip forms. This possibility has been studied in research laboratories. An example expert system in a chip form has been developed outside the automotive industry for experimentation with high-speed robot control. However, at present in the automobile manufacturing business, there are no known expert systems that have been implemented in hardware form.

Non-LISP computers are used for most small and some medium-size applications. The choice of the exact hardware is generally dictated by the size of the application and by either the fact that the chosen hardware is existing and available on site or by direct cost considerations. As a result of the availability of several software packages for building expert systems on personal computers, some of the exploratory or proof-of-concept type of work for simple systems is presently done on personal computers with 512K memory range. Also, personal computers are targeted to be the major delivery vehicles in the automobile industry for large expert systems that require specialized AI machines for development. Engineering workstations are becoming increasingly popular in the automobile industry and elsewhere. These machines can be used both for development and delivery of expert systems. Most of these workstations are based on standard commercially available 32-bit processors. Minicomputers and mainframes have been used for AI work only in situations where such computers are already in existence and where AI hardware could not be acquired. Despite the ready availability of existing mainframe systems and minicomputers, it does not appear that much AI work will be performed on those computers by the automobile manufacturers unless new, efficient packages become available.

Finally, no matter what type of computer hardware is used for development of expert systems, the delivery computer for fielding the system on the plant floor must be rugged enough to withstand the factory environment. Similarly, the software must be extremely user friendly and robust. The program should be capable of recovering from erroneous and arbitrary inputs from the user.

LANGUAGES IN USE

Several languages have been used for developing AI programs. Two of these, LISP and PROLOG, have been the ones most commonly used. In principle, any conventional computer language can be used to develop expert systems. In fact, systems based on assembly languages, BASIC, FORTRAN, Pascal, and so forth, are in existence. However, the development effort on the part of the user is often compounded due to the nature and limitations of these conventional languages and the lack of good associated programming environments.

LISP (*list processing*), which was invented by John McCarthy at the Massachusetts Institute of Technology, is the most widely used AI language in the United States. It is based on the lambda calculus and is an

excellent language for computing symbolic expressions. LISP features powerful recursion, does automatic memory management, and has broad, general-purpose use in developing AI applications. Initially, only interpreted versions of LISP were available, but today efficient and optimized LISP compilers are readily available for a wide variety of hardware environments ranging from personal computers to large mainframes and dedicated LISP machines.

PROLOG (*programming in logic*) was invented at the University of Marseille, France, by Alain Colmerauer. It has been used extensively in Europe and Japan. Like LISP, PROLOG also performs symbol manipulation. PROLOG is based on the Horn-Clause subset of classical, first-order logic and is an excellent language for logic, relational data base, and natural language applications. It features built-in backtracking, pattern matching, and unification capabilities. Like LISP, both interpreted and compiled versions of PROLOG are now readily available.

Porting LISP and PROLOG programs to different hardware environments has turned out to be a difficult task in the automobile industry. Also, the speeds of execution of programs written in those languages are generally slower than programs written in conventional languages. Therefore, for portability and speed considerations, there has been a move toward delivering and even developing expert systems in conventional languages. Most notable of such languages is the C programming language. It was developed by Dennis Ritchie at the Bell Laboratories. C was designed for and implemented on the UNIX operating system, which has lately gained tremendous popularity both in the academic and industrial environments. C provides access to low-level operations, which were traditionally done at the assembly language level. The conciseness of the C language, its portability, and faster execution speeds have led to increased interest in the language in the commercial AI market.

In the automobile industry, it is the tools based on these languages rather than the languages themselves that have found widespread use. These tools enable users in industry to do rapid development without the need to master the intricacies of the underlying computer languages. However, the AI languages do find significant use in the R&D arms of the automobile companies. These languages may be used to enhance existing applications and to explore applications of knowledge engineering to areas where commercially available tools are not effective or appropriate.

SOFTWARE TOOLS

To make the job of developing expert systems easy, AI system developers in the automobile industry generally use one of the several commercially available expert building tools or shells. These tools or shells range widely in terms of cost, functionality, performance, capacity, and other parameters of interest to an AI programmer. Dozens of such software tools are available in the market today. The simplest of the AI tools can be purchased for a few hundred dollars, whereas the more sophisticated ones cost up to tens of thousands of dollars. Due to the complex matrix of factors influencing the selection of an appropriate tool, the process of tool selection itself becomes an important task in the overall venture of an AI system development. The entire

spectrum of the AI software tools market can be divided into three broad groups: small-scale, medium-scale, and large-scale tools. As a result of the diversity of applications, tools from each of the three groups have been in use in the automobile industry.

The small-scale tools, though inexpensive and easy to use, are very limited in functionality. Also, good documentation, customer support, and training are not always available or are limited, at best. In the automobile industry, such tools are used mainly for exploratory purposes. Once the initial knowledge has been coded and the problem found to be amenable to the selected representation and inferencing techniques, the project is moved to either a mid-size or a large-scale tool. Another important use of small-scale tools has been in introducing the concepts of AI programming to the potential users in the automobile manufacturing community. These packages are ideal for this purpose because of their low cost and the fact that almost all such packages run on already existing and available personal computers.

Medium-scale tools are also used mostly on personal computers. A few of these tools run on engineering workstations and minicomputers as well. The underlying language for such tools is comprised of both conventional languages, such as BASIC, C, FORTRAN, Pascal, and assembly, and AI languages, such as LISP and PROLOG. A large number of applications in the automobile manufacturing domain can be coded using these medium-scale tools. Most of these tools employ rule-based representation and permit one or both directions of chaining. There is adequate facility for separation of control knowledge and domain knowledge, which is important in retaining the modularity and flexibility of an expert system. These tools are relatively easy to learn, and in most cases the expert system developer is not required to know the underlying language.

Medium-scale tools, though adequate for many applications, lack the flexibility and power that may be necessary for large, complex, and ordinarily intractable applications. It is mostly in these difficult automotive applications that expert systems show promise of greatest monetary returns. State-of-the-art, large-scale hybrid tools have been used to code problems of this kind. These tools have been traditionally written in AI languages, specifically in LISP. Development of expert systems using these tools is therefore usually performed on specialized, dedicated LISP machines. Some of the vendors of these tools have moved, or are in the process of moving, the underlying code into conventional languages, specifically into C. With this change, large-scale tools are now becoming capable of running on engineering workstations as well. The choice of the underlying language and the appropriateness of the development hardware have been the subject of much debate. The automobile industry, like most of the rest of the AI community, has yet to examine firsthand the pros and cons of a shift from LISP to conventional languages.

Large-scale tools are very expensive and are therefore used only in situations where other tools fail to supply the general functionality or fail to provide some specific features necessary to the implementation of the automobile manufacturing application. Usually, specialized training is required to learn the use of such tools. Working knowledge can be acquired in a matter of weeks, but the ability to write code optimized for speed of execution and fully utilize the power of these tools may take months.

Large-scale tools provide combined rule-based and frame-based knowledge representation schemes. For the rule portion of the knowledge base, a combination of forward and backward chaining is provided. The direction of inferencing can be altered at the user's discretion. These tools provide excellent graphics user interface and graphics editors, in addition to standard full-screen editors with debugging and tracing facilities. Facilities for incremental compilation of individual knowledge base elements are also provided. Large-scale tools feature the unique ability to simultaneously handle multiple alternative scenarios. Also, such tools provide facilities to perform automatic retraction and backtracking of events during the course of an expert system consultation. These and other finer features have made large-scale tools attractive for a variety of automobile manufacturing applications.

The automotive community is not just interested in general expert-system-building shells but in application-specific shells as well. These shells are tailored to specific applications and have a significant amount of domain-specific knowledge already built into them. Therefore, they simplify the task of knowledge programming considerably. Depending on the complexity of the problem being addressed, such application shells could possibly be used directly by the domain experts in the automotive industry with minimal or no assistance from knowledge engineers.

TECHNOLOGY INTERNALS

An expert system is made up of two principal components, a knowledge base and an inference engine operating on that knowledge base. The knowledge base contains knowledge about the specific domain for which the expert system is applicable. The inference engine, through dynamic interaction with the knowledge base, establishes appropriate connections across the knowledge base elements and helps answer questions or solve specific problems to which the expert system is subjected.

The inference engine is usually generic in nature and is decoupled from the knowledge base. This property allows the same inference engine to be used on several different knowledge bases. The inference engines themselves are made up of several subcomponents, such as the chaining, searching, reasoning, and tracking mechanisms. By appropriate choice of combinations of the above types of mechanisms and suitable adjustment of other finer features, a large variety of general-purpose inference engines can be obtained.

In a rule-based expert system, the multitude of rules within the knowledge base are connected together dynamically by the inference engine through a process called "chaining." There are two types of chaining mechanisms, forward chaining and backward chaining. Most inference engines have one principal direction of chaining with only minor support for the other direction. Large, powerful inference engines permit chaining in both directions at the user's discretion. Such chaining mechanisms are referred to as "bidirectional." As a result of the diversity of applications in the automobile industry, ranging from very small and simple to very large and complex, all three types of chaining mechanisms—forward, backward, and bidirectional—are useful.

While operating on any given knowledge base, the inference engine has to perform several searches to find the desired pieces of information. Hence, clever choices of search mechanisms can greatly speed up the execution time of an expert system. Several standard search algorithms are available. "Tree-traversal" algorithms, such as the "depth-first" and the "breadth-first" algorithms are most popular. Assuming that the search space is structured like a tree, the depth-first algorithm searches through the entire space by picking up one branch of the tree at a time and traversing deep through the nodes until a solution is obtained or the "leaf nodes" of each branch are reached. The breadth-first mechanism, on the other hand, first traverses the first nodes of each branch, then the second-level nodes, and so on, until the final leaf nodes are reached or the solution is obtained.

Except for the preliminary strategic decision regarding direction of searching, both depth-first and breadth-first mechanisms are essentially brute force methods. Several other algorithms that improve upon these methods have been developed. These usually employ simple optimization procedures using cost functions. Examples include the "branch and bound" and the "A*" algorithms. Game-theory concepts have also been utilized to refine the search process of an inference engine. "Alpha-beta" pruning and "minimax" approaches are examples of application of such concepts. Almost all of the expert systems used in the automobile industry today employ either the depth-first or the breadth-first search or, at best, a user controlled combination of the two.

The reasoning process of an inference engine can be of either "monotonic" form or "nonmonotonic" form. In monotonic reasoning, once the truth or the falsity of any attribute within the expert system is set, it cannot be altered or reset. This poses a problem in situations where the application calls for an examination of hypothetical or alternative scenarios. Non-monotonic inferencing permits changing values of attributes. Also, different values can be assigned simultaneously to the same attribute with multiple "worlds" in nonmonotonic reasoning. The process of keeping track of different attributes and their associated values in a nonmonotonic system is accomplished through the use of a "truth-maintenance" subsystem. Most of the expert systems in use today use only monotonic reasoning, though some limited use of nonmonotonic reasoning is fast becoming prevalent.

To keep track of the changes that may occur during the execution of an expert system program, the inference engine uses either "chronological" or "nonchronological" tracking methods. Chronological methods are a crude form of keeping track of the effects of altering the previously defined or internally determined facts in an expert system. Additions of new facts to the system are considered as "events in time," and these facts are chronologically backtracked if a later event attempts to modify a prior event. In nonchronological tracking, the interdependencies of various events are used to prune through the fact base. So only those prior events that are directly or indirectly dependent on a new violating event are examined and modified. This helps in efficient maintenance of the fact base integrity.

The knowledge base portion of the expert system contains knowledge that is specific to the domain of application. In most automotive applica-

RULES
<p>IF: Car has problems and car stalls or car has sluggish-cranking</p> <p>THEN: Remedial action is charge battery (CF=0.7) or replace spark plugs (CF=0.4)</p>

FIGURE 3 Rule-based approach.

tions, this knowledge is usually obtained through knowledge acquisition sessions with the domain expert. It is then coded by the knowledge engineers using one of the standard knowledge representation formats. Rule-based, frame-based, and logic-based formats are the most commonly used ways of representing knowledge in an expert system. Choice of representation is generally a function of the nature of knowledge that needs to be stored. Presently, the automobile industry uses primarily rule- and frame-based representations.

Rule-based systems represent knowledge in the standard IF-THEN format. In a forward chaining system, if the premise conditions of rules are satisfied, then the conclusion portions are automatically asserted to be true. In a backward chaining system, queries are made to the system. In response, those rules whose conclusion portions match the queries are activated. The use of "certainty factors" (CFs), which enable the user to get a feel for the confidence level for any solution, is a valuable feature of rule-based systems. These CFs are not cumulative probabilities, and, therefore, the sum of CFs of a set of recommended solutions will not generally be unity. An example of a rule pertaining to automobiles is shown in Figure 3.

Frame-based systems make use of the "object-oriented" programming style and are becoming increasingly popular in the AI community. A frame represents all of the attributes of an object in the knowledge base. These frames interact with each other through a process of "message passing." As a result of the hierarchical arrangement of frames in a tree-like structure, inheritance of properties from parent objects to descendant members is greatly facilitated. An example of the same scenario used in Figure 3 is shown in frame-based representation in Figure 4.

Logic representation is a more formal method of representing domain knowledge in an expert system. This method makes use of the standard logical operators, such as conjunction, disjunction, and negation. In addition, both existential and universal quantifiers are used to appropriately scope the variables in logical expressions. Several variations, such as propositional, first-order, and higher-order logics are used depending on the need and complexity of the application. A sample logic-based representation of the automobile example is shown in Figure 5.

FRAMES	
OBJECT:	Car
SLOT:	Inheritance (override)
SLOT:	Has (battery) (spark plugs)
SLOT:	Symptoms (stalls) (sluggish cranking)
SLOT:	Remedial action (charge battery) (replace spark plugs)

FIGURE 4 Frame-based approach.

SKILLS NEEDED

The job of a knowledge engineer (KE) in the automobile industry usually encompasses a large variety of domains and involves close interactions with human experts. Accordingly, a KE must have excellent general analytical capabilities and good interpersonal skills. For manufacturing applications, a background in engineering forms an ideal preparation. Earlier, professionals with data processing or computer science backgrounds were considered most suitable for knowledge engineering jobs. More recently, the emphasis has shifted toward KEs with only peripheral knowledge of computer science but good knowledge of the associated field, be it engineering, finance, marketing, or any other activity.

In general, a KE has two distinct job responsibilities: knowledge acquisition and knowledge coding. During knowledge acquisition, the KE interacts with a human expert and acquires or extracts the expert's knowledge. This extracted knowledge is then coded during the knowledge coding process. It has been long debated whether the expert should do the coding

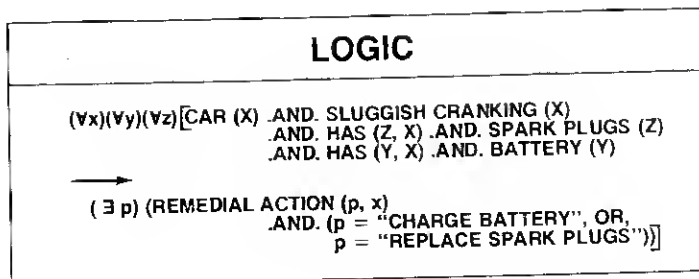


FIGURE 5 Logic-based approach.

directly, thereby eliminating the need for the KE and knowledge acquisition. This philosophy has other advantages and disadvantages, and no consensus has been reached on it in the AI community. The skills for knowledge coding are usually acquired through college preparation, but the knowledge acquisition skills are largely dependent on the nature of the person (inquisitive, easy to get along with, etc.). These skills come with time, though there are courses offered by commercial companies that assist in gaining the necessary skills for adequate knowledge acquisition work. Most companies that sell expert systems building software packages also provide good training on the use of these packages. It is best to undergo the prescribed training in order to better understand the intricacies of these packages.

The following is a representative sample of a technical background for a KE in the automobile manufacturing environment. An engineering background, preferably mechanical, industrial, or electrical, is ideal. Knowledge of computers, including experience with minicomputers and mainframe systems, and proficiency in at least one conventional language and one AI language, preferably LISP, is essential. Knowledge of microprocessors and computer architecture is helpful. Advanced mathematics, including game theory, probability, calculus, modern mathematics, logic theorems, and numerical analysis is recommended. Exposure to arts/humanities is useful. Psychology, linguistics, philosophy, and cognitive sciences are generally considered good for a well-rounded KE.

In the automobile manufacturing business, the manufacturing expertise almost always lies with personnel who have had several years of hands-on experience. These people, though otherwise very qualified, often have little or no experience with any form of computer programming, much less any knowledge of AI programming. The burden of translating the expertise of these persons into an expert system lies solely with the KEs. Therefore, a KE in the automotive business must have a sound technical expertise to understand the language of the domain and, in addition, must possess excellent interpersonal communication skills to extract the knowledge from the expert(s).

PRODUCTS AND SERVICES FROM EXTERNAL COMPANIES

Because the technology is new and the experience and skills needed are not easily found within the automobile industry, there has been reliance on external AI companies. Major automobile manufacturers have made significant investments in leading AI companies. These affiliations have enabled the auto manufacturers to launch and implement AI projects at an accelerated pace. Such relationships lead to two principal benefits. First, they provide access to the technology know-how and the ability to tailor the products and services of the AI companies to the needs of the automakers. Second, large projects can be contracted out to the affiliated companies for expedient completion whenever the expertise or the staff is not internally available. Other automakers, who may not have established strategic relationships with AI companies, also have access to and utilize such external resources.

Manufacturing applications form about two thirds of the entire expert systems market. The automotive and related industries constitute a very significant portion of the entire manufacturing industry. Accordingly, AI companies are paying close attention to the automobile manufacturers. There is now a trend among AI software development companies to develop specialized tools targeted for use in the manufacturing applications. This is somewhat of a departure from the earlier trend of focusing on large, general-purpose tools. The hardware vendors have also paid special attention to the needs of the automobile manufacturers. Rugged, relatively low-cost AI hardware, fit for use on the plant floor, is now becoming available in the market.

In the past, the automobile industry has had to rely primarily on technical consulting services of major AI companies. However, this dependence continues to decrease as more and more in-house expertise becomes available.

RELIABILITY, VALIDATION, AND ACCEPTANCE

The issues of reliability, validation, and acceptance of expert systems have been of concern since the very first systems were built. As the automotive industry makes a transition from development to deployment of expert systems, the focus on these issues will become increasingly evident.

The question of reliability is of primary importance. The computer hardware should be rugged enough to withstand the rough factory environment, and the software must be sufficiently robust to gracefully recover from spurious inputs and any line noise. Once the hardware and software are established to be reliable in this regard, the reliability of the knowledge contained in the expert system has to be examined. Before any expert system can be placed on the plant floor, it must be established that its recommendations or actions are accurate and reliable. Of course, the extent to which a system needs to be accurate or reliable is a function of how critical the job being performed is and what consequences an erroneous output may have. The monetary losses due to misdiagnosis, poor designs, inefficient scheduling, inadequate or excessive inventories, and so forth, can be significant. In fact, most of the expert systems being developed today in the automobile industry are initiated on the expectation of large financial returns. These returns are possible only through reliable, consistent, and accurate systems.

Validation involves certification that the system has been tested and has proven capable of performing the task on the targeted application domain. Depending on the nature of application, validation may be a straightforward procedure or an extremely complicated and difficult task. In the automobile manufacturing environment, the expert, or a select group of end users, validates the system by running several cycles of it on some simple, typical cases, as well as on previously solved difficult cases. A method of extended validation can be used in conjunction with the previous method. This involves testing on an assortment of real, existing, previously unsolved problems. Validation becomes a problem only if the area of application selected is new and no good representative sample cases are

available for testing. In such a situation, synthetic cases can be developed and the system tested against those cases. For large, critical systems, a combination of real and synthetic cases is generally used. System refinements are usually performed in parallel with validation to remove any shortcomings and to fill in any detected gaps in knowledge.

Only a few documented systems are in place and operational in the manufacturing environment within the automobile industry. The fact that these systems are planned to assist plant personnel, not replace them, has been crucial in promoting acceptance of expert systems in the automobile industry. Another reason for initial acceptance has been the sense of ownership instilled among the users of those systems. The industry is being careful in providing all necessary training that may be needed to use, maintain, and update those systems.

TECHNOLOGY ISSUES

The issues facing AI technology in the automobile industry can be broadly classified into three levels: implementation, theoretical aspects, and research. The implementation issues are getting the maximum attention in the automobile industry because those issues affect the industry most directly. The more pragmatic portions of the theoretical and research issues are being studied by the research divisions of the automotive companies, whereas the esoteric portions are being relegated to the academic and other research institutions.

In the implementation category, cost has been the foremost issue. Due to the scarcity of qualified AI personnel, the industry has had to rely heavily on external companies that specialize in the AI business. This has added to the total costs of implementing the projects. Besides the costs of training the internal personnel and buying expensive computer hardware for development work, the costs and availability of adequate hardware for fielding the systems on the plant floor for day-to-day use have been a serious issue. This problem has lately been somewhat alleviated with reductions in the cost of the so-called "delivery computers" and the ability to port large development systems written in AI languages into condensed, small delivery systems in conventional languages.

There is such a tremendous diversity of mechanical and electronic equipment in the automobile manufacturing plants that integration of AI systems poses a significant challenge. There are large amounts of data stored and maintained on conventional mainframe computers resident in the manufacturing plants. Communication and efficient data transfer/retrieval between these data base computers and the computers hosting the knowledge systems is a major task in the actual fielding and usage of AI systems. The computer industry is making efforts to assist in this work by providing standardized communications and networking packages.

Another important issue that has hindered greater proliferation of AI activity is the lack of availability of committed experts. Ironically, perhaps each of the manufacturing plants in the automobile industry has at least one resident expert in the associated domain of activity. So the problem is not with the lack of existence of expertise but with the committed availability of the same.

Awareness of the AI technology within the automobile industry must also be addressed. Increasing awareness includes providing concise, accurate information relating to the appropriateness and capabilities of the technology so that the user community has realistic expectations. Concerted efforts are under way throughout the industry to make people aware of the benefits of the technology, particularly in manufacturing applications.

In the theoretical aspects, focused effort is needed to specialize the control structures, search techniques, and representation methods to the automobile manufacturing domain. Some work has already begun in that direction. Similarly, effort is needed to develop good methods for spatial reasoning because many activities in the automotive industry involve reasoning about three-dimensional shapes, contours, and locations. Due to the voluminous, yet incomplete, nature of data often encountered in automobile manufacturing applications, large systems will have to use better and more efficient mechanisms to ensure complete automatic redundancy/consistency checks.

Finally, there are some research issues that will benefit the automobile industry when they are resolved. First and foremost is the speed issue. The speed of response of most systems in use today is unacceptably slow for continuous on-line operations. It is therefore no surprise that almost all of the AI applications in the automobile industry are off-line in nature. Research effort is under way to improve both the computer hardware design and software programs to increase execution speeds. Parallel processing hardware shows promise, but so far no concrete use of it has been made in the automotive business.

Automatic learning is another area of research. There are no documented systems in the automobile industry that can learn from previous mistakes or experience in any significant way. The ability to learn would be extremely useful, especially in the flexible manufacturing environment toward which the automobile industry is moving today. Similarly, automatic programming based on simple, generic-type specifications is under heavy research. Advances in this area would be of significant value to the automobile manufacturing business.

PROMISE OF THE FUTURE

What the automobile industry has experienced is just the beginnings of the influence of AI in manufacturing. There is tremendous diversity of manufacturing applications, such as machining operations, diagnosing of electrical and mechanical components, general troubleshooting, selection, scheduling, controllers, robotics, and machine vision. The potential for use of expert systems to assist in each of the numerous applications that come under the umbrella of automobile manufacturing is immense. So far, a large number of applications in the manufacturing arena have been in the domain of diagnostics because it is among the better understood application areas within AI technology. However, the automobile manufacturing business stretches far beyond the mere boundaries of diagnostics. As AI technology advances and the paybacks of its use become evident, proliferation in all areas of manufacturing is expected to result.

Even as the giant automobile industry continues to explore the possibilities with emerging AI technology, there is belief and hope in a bright future for AI in automobile manufacturing.

ATUL BAJPAI
BARBARA A. SANDERS

Awareness of the AI technology within the automobile industry must also be addressed. Increasing awareness includes providing concise, accurate information relating to the appropriateness and capabilities of the technology so that the user community has realistic expectations. Concerted efforts are under way throughout the industry to make people aware of the benefits of the technology, particularly in manufacturing applications.

In the theoretical aspects, focused effort is needed to specialize the control structures, search techniques, and representation methods to the automobile manufacturing domain. Some work has already begun in that direction. Similarly, effort is needed to develop good methods for spatial reasoning because many activities in the automotive industry involve reasoning about three-dimensional shapes, contours, and locations. Due to the voluminous, yet incomplete, nature of data often encountered in automobile manufacturing applications, large systems will have to use better and more efficient mechanisms to ensure complete automatic redundancy/consistency checks.

Finally, there are some research issues that will benefit the automobile industry when they are resolved. First and foremost is the speed issue. The speed of response of most systems in use today is unacceptably slow for continuous on-line operations. It is therefore no surprise that almost all of the AI applications in the automobile industry are off-line in nature. Research effort is under way to improve both the computer hardware design and software programs to increase execution speeds. Parallel processing hardware shows promise, but so far no concrete use of it has been made in the automotive business.

Automatic learning is another area of research. There are no documented systems in the automobile industry that can learn from previous mistakes or experience in any significant way. The ability to learn would be extremely useful, especially in the flexible manufacturing environment toward which the automobile industry is moving today. Similarly, automatic programming based on simple, generic-type specifications is under heavy research. Advances in this area would be of significant value to the automobile manufacturing business.

PROMISE OF THE FUTURE

What the automobile industry has experienced is just the beginnings of the influence of AI in manufacturing. There is tremendous diversity of manufacturing applications, such as machining operations, diagnosing of electrical and mechanical components, general troubleshooting, selection, scheduling, controllers, robotics, and machine vision. The potential for use of expert systems to assist in each of the numerous applications that come under the umbrella of automobile manufacturing is immense. So far, a large number of applications in the manufacturing arena have been in the domain of diagnostics because it is among the better understood application areas within AI technology. However, the automobile manufacturing business stretches far beyond the mere boundaries of diagnostics. As AI technology advances and the paybacks of its use become evident, proliferation in all areas of manufacturing is expected to result.

Even as the giant automobile industry continues to explore the possibilities with emerging AI technology, there is belief and hope in a bright future for AI in automobile manufacturing.

ATUL BAJPAI
BARBARA A. SANDERS

ARTIFICIAL INTELLIGENCE APPLICATIONS IN CHEMISTRY

INTRODUCTION

What is AI?

Artificial intelligence (AI) is a field of computer science encompassing programs that simulate human intelligence. A definition of human intelligence must include the intrinsic "know-how" that enables people to solve problems using intuitive and deductive methods. Computer scientists working in AI are constantly challenged to produce applications that better mimic human thought processes. This means that AI is constantly changing as those tasks that are attempted become more difficult. Because people do not behave in easily described ways, they continue to possess capabilities that computers do not.

AI researches are investigating a number of key "human" areas where computers may someday simulate people's actions and behavior. These areas include

1. **Learning:** People's decisions and subsequent actions are often governed by their past experiences. Deductions formed by these experiences serve as reasons for making decisions. Researchers are trying to determine whether machines can also "learn" from previous trials and make decisions based on the results of the trials.
2. **Vision:** Seeing objects helps people make decisions about these objects. By allowing computers to "see" images, through digitized photographs or video displays, they may be able to make decisions on an object's spatial relationship and makeup. AI applications used by the Defense Department and other manufacturers would benefit from this technology.
3. **Problem solving:** Researchers are programming computers to solve problems based on defined rules and strategies (or algorithms). Examples of this problem-solving capability are chess programs, which exhibit championship-caliber play, and symbolic mathematics programs, which solve algebraic and calculus problems.
4. **Logic:** As with problem-solving studies, researchers are looking into the possibility of using computers to solve logic problems based on the fundamental principles of a field of study. Most of this work has centered on using mathematical rules.
5. **Robotics:** Current research into robotics attempts to program robots for specific tasks within a changing environment (i.e., different actions may be required to perform the same task). The use of robotics in chemistry is, almost exclusively, simple mechanical arms that are capable of performing preprogrammed, repetitive movements. Although these actions may be complicated, tedious,

exacting, cost effective, or may include vision systems, they will not be considered AI in this article.

6. Natural language: Human-to-computer interaction has evolved through research into natural language use. Researchers are now working to allow computers to recognize English language sentences as valid program input. When this capability is exploited, people can use their natural language to run computer programs. For example, a language translation program may interpret English sentences into French ones, and vice versa. This article expands the scope of natural language studies to include representing, manipulating, and presenting chemical structures.
7. Expert systems: Researchers studying expert systems are writing programs that mimic thought processes used by experts in solving problems in their field. These programs are based on rules of thumb that experts develop through years of experience. The problems solved are often vague, complex, or may include incomplete or inexact information.

AI has developed a vocabulary of its own over the past 30 years. Appendix A contains a glossary of AI-related terms used in this text.

Expert Systems

AI applications that are expert systems contain three basic pieces: a knowledge base, an inference engine, and a user interface. The knowledge base holds the facts on which the program reaches decisions. The inference engine instructs the program how to use its knowledge base to reach these decisions. The user interface is the method that the program and chemist use to communicate with each other. These three sections of an expert system are described below.

Knowledge Base

An expert system's knowledge base is a resource of information about a specific problem area. How programmers encode this information in an understandable and usable way is one of many areas of research in AI. Some of the more common ways of encoding the knowledge base are illustrated in Figure 1 and are described below.

Production Rules: Production rules present facts that are dependent on the truth of other facts or hypotheses. These rules take the form IF x is true THEN y is true. This simplified case may be expanded to include multiple hypotheses and confidence values associated with the hypotheses.

Frames: A frame is a data structure that incorporates hierarchical data dependencies between objects. In a frame, the upper level (or parent) objects pass on attributes to objects beneath them (children) in the hierarchy (children inherit attributes from their parents). One example of a frame is the classification of animals in taxonomy.

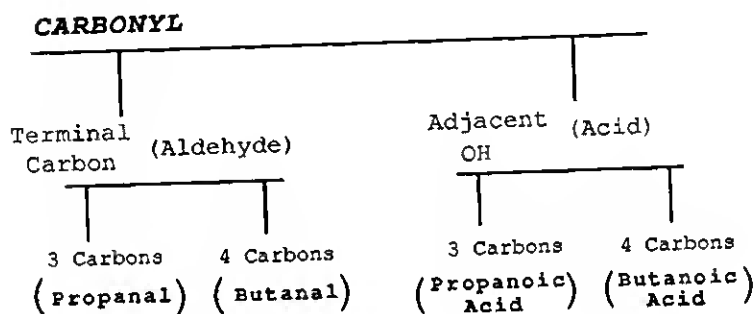
Networks: Networks establish links (or associations) between objects by storing them with connections based on real-world traits. For example, the three objects CAT, DOG, and MAMMAL may be connected by a trait "IS-A," which links MAMMAL to both DOG and CAT. In addition, "ENEMY-OF" may link DOG to CAT directly.

PRODUCTION RULES

IF **Carbon X contains $C = O$** and
 Carbon X is a terminal carbon

THEN **Functional Group is Aldehyde**

FRAMES



NETWORKS

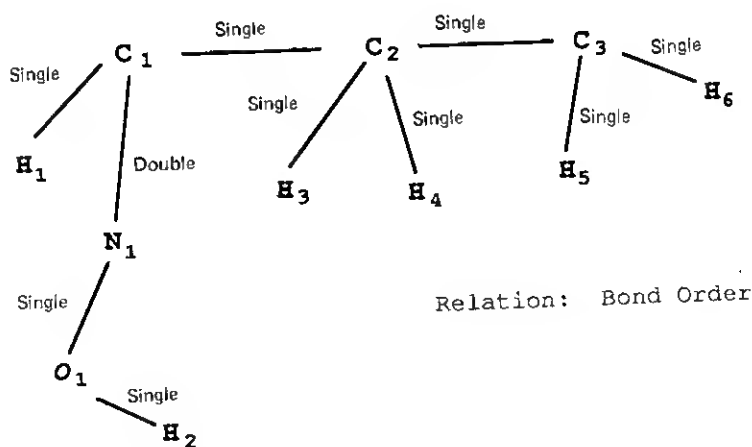


FIGURE 1 Encoding the knowledge base.

Inference Engine

Inference engines are central programs that manipulate rules and facts in the knowledge base so that conclusions can be reached. The program's structure depends strongly on the type of knowledge base that the expert system incorporates. Inherent in most of the program structures chosen, however, is a basic set of functions that expert systems perform. These functions are described below by using a knowledge base that incorporates production rules. This choice was made because most commercially available expert systems incorporate production rules.

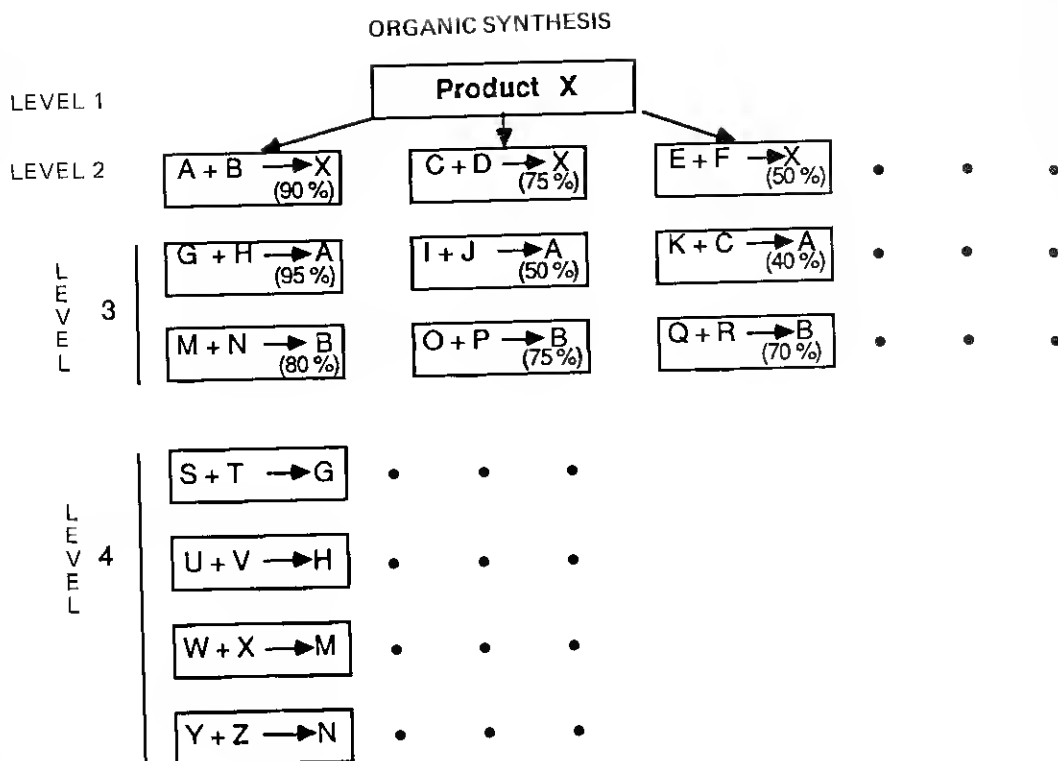
A second consideration for the program's structure is how the user wants to solve a problem. If a user begins with several hypotheses and wants to determine whether any of these hypotheses are correct, then the program must examine all the facts using a goal-directed approach. However, if the user begins with a series of facts that are known to be true and wants to determine what conclusions can be reached, the program must use a data-directed approach.

A goal-directed expert system begins with a set of possible hypotheses and searches the knowledge base until one hypothesis is proved. Under this type of expert system, a reverse-chaining mechanism is used to search the knowledge base to find a rule that concludes the initial hypothesis. The IF clauses from the starting rules become the hypotheses for the next level of the search. The process continues until all of the remaining IF clauses are known to be true (hypothesis is true) or until no more rules apply (hypothesis is false). An example of this mechanism is illustrated by an organic synthesis problem:

A scientist wants to formulate chemical X. The expert system searches its rules (in this case a list of reactions) for a reaction that yields X as a product (the conclusion). The expert system looks at the reactants (the IF clause) and searches the remaining rules for reactions that yield each reactant. This process continues until simplified reactions are found that meet the needs of the scientist (Fig. 2).

Data-directed expert systems begin with a list of facts and attempt to draw conclusions on the basis of these facts. As with goal-directed systems, data-directed systems begin solving the problem by searching the knowledge base for relevant rules. These searches are conducted using a forward-chaining mechanism. When all the beginning facts are found in the IF clause of a rule, the system adds the THEN clauses from the rule to the list of known facts. The program must then search the knowledge base to find rules that can use the new fact. When expert systems use forward-chaining methods, all of the rules in the knowledge base are searched until all of the relevant rules are used. An example of the use of forward chaining is illustrated by an infrared (IR) and mass spectrum (MS) identification problem:

A scientist wants to identify a compound for which the peak locations in IR and MS are known. The program searches the knowledge base for a rule that has these peak locations in its IF clause. Using the conclusions in the THEN clause of this rule, the system continues to search its knowledge base for rules that will identify probable structures of the compound.



METARULE: IF: Two reactions have the same product
 THEN: Choose the reaction with the highest yield

FIGURE 2 Expert system search.

The two examples illustrated above show that the structure of a problem determines what type of search to use. In the first example, the scientists knew the conclusion but wanted to find the beginning hypothesis. A data-driven approach to this problem would be to determine all possible reactions of all available compounds and all possible reactions of the new compounds until the desired product was produced. Obviously, the problem demands a goal-directed approach.

In the second example, the scientist began with a set of facts and was looking for a conclusion. The goal-directed approach to this problem would be to attempt to prove whether each one of the millions of compounds reported in the chemical literature could be responsible for the observed spectrum. Again, the structure of the problem determines the structure of the expert system.

The performance of an expert system may be increased by using heuristic rules to eliminate possible but unlikely solutions, given the constraints of the problem. Heuristics may be incorporated by using metaknowledge. In knowledge bases that use production rules, metaknowledge takes the form of metarules. These rules instruct the expert system how to choose

between multiple rules that are relevant. Relating back to the organic synthesis example (Fig. 2), a metarule may take the form

IF more than two reactions are available (as conclusions)
THEN use the one with a higher product yield.

An expert system's performance may also be increased by combining the use of forward and backward chaining. The system may apply a forward-chaining approach on some background information in an attempt to eliminate hypotheses. This approach may also order the possible hypotheses by the probability of their occurrence.

Searching patterns also affect the performance of an expert system. Primarily, two types are used in expert systems: depth-first and breadth-first searches. Using the organic synthesis problem from above, a depth-first search would take the "best" reaction to form the product and try to find the "best" reaction to make each starting material. This search limits the number of reactions to check by eliminating materials that are made from reactions different from the best one. A breadth-first search would investigate all possible reactions that yield a particular product. The choice between these two types of searches depends on the problem being solved by the expert system.

User Interface

How an expert system communicates with its user is defined by the program's user interface. An expert system's user interface should provide the following functions: a method for an expert to enter knowledge (rules), a method to modify rules, and a presentation system that allows novice users to work with the system effectively. The contents of the expert system's knowledge base must be expandable by some mechanism. When an expert adds knowledge, or rules, the program must check for contradictions in information. It should also test these new rules against previously defined problems. In addition to adding rules, the system must allow an expert to modify rules so that more correct conclusions may be reached by the system.

The system should provide ways for the novice user to execute his/her searches effectively. These methods may take the form of prompting for information, explaining why the information is needed, and telling how a conclusion is reached. Explanations usually display the progression of rules used by the system to solve the problem. Some of the more sophisticated commercial expert systems take advantage of computer graphics to perform these functions.

Appendix B contains a list of chemical applications of expert systems, some of which are described in more detail in this article.

Natural Language

Natural language AI applications are those applications in which the computer has an ability to communicate in a "normal" English fashion. In chemical applications, the concept is generalized to the computer's ability to interact with chemists using the vocabulary common to chemists. Another use of natural language has the computer recognizing molecules, molecular substructures, and chemical reactions. There are also "grammars" built to

describe chemical reactions. In this use of natural language, a sentence is a description of a chemical reaction. Most natural language applications in chemistry revolve around understanding the domain of chemistry and interacting in English with a chemist at a computer terminal. The computer language for understanding molecules is related more to symbolic computation than to traditional English language applications. The symbolic computation of chemical structures is also the most successful.

HISTORY OF AI APPLICATIONS IN CHEMISTRY

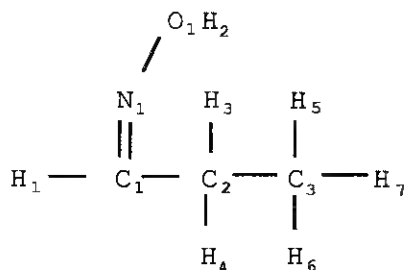
Chemistry was one of the first disciplines, aside from computer science, to actively engage in research on AI techniques. The first chemistry AI project was the DENDRAL project at Stanford University. It began in 1964, involved more than 50 researchers, and produced more than 100 articles and two books. The original goal of the DENDRAL project was the automatic interpretation of MS data. However, during its 20-year history, it has also investigated a variety of other topics, including automated learning, computerized representation of chemical structures, applications of graph theory to chemical problems, exhaustive generation of chemical structures, and proton and C^{13} nuclear magnetic resonance (NMR) interpretation. A detailed account of the project can be found in Refs. 1 and 2.

Another group of closely related areas of investigation that began just after the DENDRAL project are learning machines, structure-activity predictions, and pattern recognition. At the time (the late 1960s) they were considered by many to be AI applications. These techniques are based on multidimensional geometry; each dimension corresponds to a different physical property. Conclusions are drawn about the compounds and their properties, on the basis of their spatial arrangement in the multidimensional space. When confined to limited sets of compounds (usually either closely related or structurally simple), these approaches yield useful results. The limited ability of these techniques to resolve complex real-world problems was another driving force to develop the AI techniques described in this article. Much of the underlying chemical principles learned in these investigations are applicable to AI.

Work in applying AI technologies to chemistry has recently expanded beyond the traditional academic environment. This trend is being driven by two major factors: the highly publicized successes of several expert systems and the decrease in the cost of computer hardware and AI software development tools. Although academia continues to develop new techniques, the chemical industry has begun to apply older, more developed techniques to solve their problems. Vendors are using AI techniques to enhance existing products and develop new ones. Other industrial researchers are developing proprietary systems in an attempt to gain competitive advantage.

REPRESENTING CHEMICAL STRUCTURES USING COMPUTERS

Natural language applications resolve into two classes: the language of chemical structures, substructures, and reactions and the methods used to convert between that language and English sentences. Chemical structure language is based on algorithms for representing molecules in the computer and a set of syntax rules for manipulating those representations [3, 4] (Fig. 3).



PICTORIAL

	C1	C2	C3	N1	O1	H1	H2	H3	H4	H5	H6	H7
C1	0	1		2		1						
C2	1	0	1					1	1			
C3		1	0							1	1	1
N1	2			0	1							
O1				1	0		1					
H1	1					0						
H2					1		0					
H3		1						0				
H4		1							0			
H5			1							0		
H6			1								0	
H7			1									0

CONNECTION
TABLE**Q N U 3**

U → double bond

N → hydrogen-free nitrogen

Q → hydroxyl group -OH

3 → three unbranched carbons

WISSWESSER
LINE NOTATION

(C H (N (O H)) (C H H (C H H H)))

LISP LIST

FIGURE 3 Computerized chemical structure representation.

This language requirement was originally defined by researchers who were storing chemical information in computer files. Wiswesser line notation [5, 6] and its derivatives were developed to uniquely define chemical structures as a string of characters. Each character represents a specific fragment of a molecule, allowing the computer to "recognize" and manipulate a molecule.

An alternative approach to chemical language uses graph theory, defining molecules as vertices and the connections between them [7]. This approach creates a connection table (Fig. 3) or matrix, whose rows and columns refer to atoms. The values stored in the matrix describe the type of connection between atoms. Syntax rules can be defined for manipulating these computer representations, allowing substructures of molecules to be defined and matched. Computer representations of molecules have created an industry. Today, large data bases of chemical structures are available commercially [8]. Molecules may be drawn on a computer screen and the data base searched for "substructures," or chemical reactions, which include the molecule. The popularity of these data bases is causing the chemical literature of the past to be computerized, thereby increasing the amount of available chemical knowledge.

The second use of natural language techniques in chemistry is more familiar to computer scientists. Natural language systems attempt to understand English sentences that contain chemical information. These systems function as user-friendly interfaces to other chemical expert systems or as intelligent interfaces to chemical data bases. Understanding English-phrased questions about data bases is a simpler problem than understanding problems relating to chemistry. Commercial natural language systems can perform some of these tasks in chemical applications, but several systems have been developed specifically for chemistry.

CHEMICAL STRUCTURE ELUCIDATION

Initial chemical identification techniques used time-consuming tests based on highly specific chemical reactions to determine the presence or absence of specific functional groups. Which molecular substructures were considered functional groups was partially determined by these tests. These first techniques were replaced mostly by spectroscopic techniques. Spectra of known compounds were visually compared to those of the unknown, with the spectroscopist looking for exact matches. As libraries of reference spectra grew, computers were required to perform the searches. Although library searching is straightforward, it becomes impractical as the size of the reference library increases. In addition, library searching cannot identify compounds whose spectra are not in the library, such as newly synthesized compounds. For this reason, other approaches have been developed.

Structure elucidation is a prime area for AI applications because it requires both scientific expertise and problem-solving capabilities. Although each researcher may use a different variation, the general approach given below is used to determine the structure of unknown compounds (Fig. 4).

First, one determines the molecular formula and identifies as many structural fragments (functional groups) contained in the molecule as possible. Information on molecular formula and structural fragments generally comes from spectral data but can come from any source at the chemist's disposal. Next, internal consistency checks are used to minimize the list of possible structural fragments. This is accomplished by combining data

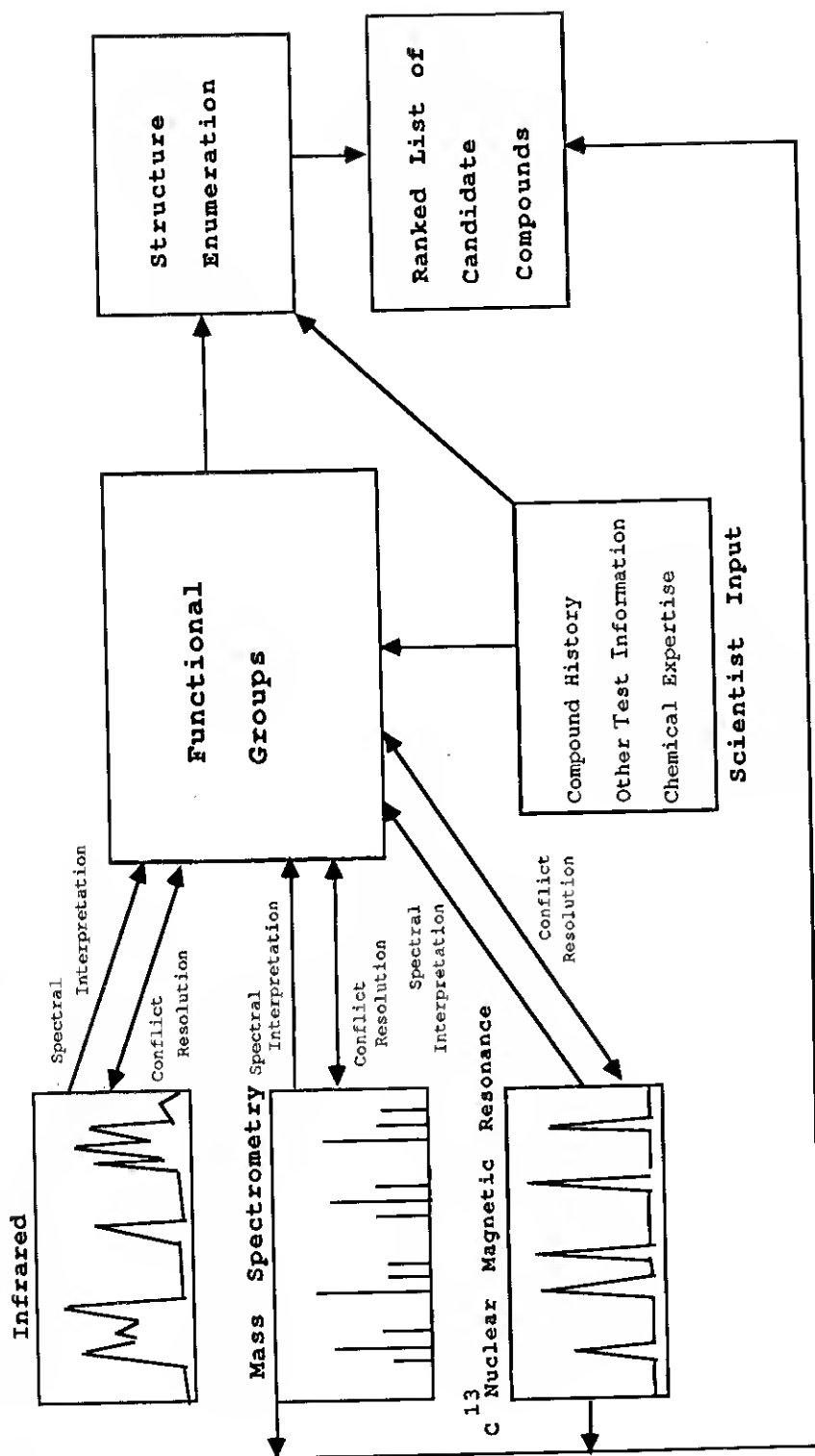


FIGURE 4 Chemical structure elucidation.

from multiple sources (e.g., IR and MS) to eliminate fragments suggested by one source but ruled out by the other. The fragments in this final list are connected together in all chemically possible ways. If the connections result in more than one structure, the structures are ranked on the probability that they are the unknown compound. Finally, the scientist must devise additional chemical tests to distinguish between the remaining candidate structures.

Programs have been written to perform each step in this process. Some perform only one step, whereas others attempt to automate the entire process. In general, these programs do not combine information from multiple spectroscopies adequately. This problem results because spectroscopists tend to be experts on only one type of spectroscopy. A second problem is resolving conflicting interpretations of data from different spectroscopies. These problems affect both the determination of structural fragments and internal consistency check portions of the problem. These areas should involve as many types of spectroscopy as possible. For that reason, determining structural fragments and performing internal consistency checks are currently the areas of most active research.

Determination of Structural Fragments

Determining the structural fragments present in an unknown molecule can be done using a variety of techniques; however, most AI research involves spectral data interpretation. Not all useful work in this area involves AI. Many programs combine library searching and pattern recognition with AI techniques to help determine structural fragments. Pattern recognition compares the unknown spectrum with a pattern that is characteristic of a particular class of compounds. Library searching can be used to search fragments of spectra that correspond to fragments of molecules. The only important aspect is to obtain a reliable list of the possible structural fragments.

AI techniques try to interpret spectra using the same interpretation rules that a spectroscopist would use. AI techniques have an advantage over spectroscopists because the same rules used by the spectroscopist are employed, and the computer system does not forget rules or bias their application. Unfortunately, it is currently impossible to encode all the knowledge possessed by spectroscopists into an AI system. AI structure elucidation systems are comparable in performance to a postgraduate spectroscopist [9]. Below, the approaches taken to interpret the various types of spectral data are described.

IR Spectroscopy

Spectroscopists have known for some time that certain functional groups and the manner in which they are attached to the molecule cause characteristic absorptions in the IR. These patterns were tabulated as early as 1950 by N. B. Colthup [10]. Pattern recognition techniques have been used to take advantage of these patterns. Early work using AI techniques attempted to computerize these tables to automatically interpret IR spectra [11,12]. These programs must be able to deal with the following problems: Many functional groups absorb at each frequency; functional groups can cause more than one absorption; the physical state of the sample can shift peak locations and intensities; and the solvents used in sample preparation and minor impurities can shift or mask peaks.

Below is an example of the type of rule that can be found in an expert system to interpret IR spectra:

- IF 1. the spectrum is condensed phase, and
 2. there is a large peak between 3100 and 3500 cm^{-1} , and
 3. there is a sharp peak at 1025 cm^{-1} ,
 4. there is little evidence for an amine
THEN there is suggestive evidence for a primary aliphatic alcohol.

Recent work has focused on reducing the task of codifying the rules required to identify the functional groups of interest. This approach extracts rules automatically from the spectra of known compounds [13].

Mass Spectroscopy

The original work on MS interpretation was done through the DENDRAL project. The first step in interpreting MS data is to determine the molecular ion (and, therefore, the molecular weight) and, using that information, determine the molecular formula. Because of the small differences between the exact masses of molecules of the same nominal molecular weight, the exact molecular weight of the compound is sufficient to uniquely determine the molecular formula (nitrogen $\rightarrow 14.0067\text{ amu}$; $\text{CH}_2 \rightarrow 12.000 + 2 [1.007825] = 14.01565\text{ amu}$). If a high-resolution spectrum is not available, the remaining fragments will often contain enough information to determine the molecular formula. Once the molecular formula is known, an upper limit is placed on the number of possible compounds. Unfortunately, that limit will be extremely large for "interesting" molecules.

Mass spec/mass spec (MS/MS) is a recent instrumental development that can take the MS of each peak in the original MS. This allows the analyst to obtain very detailed information about a small portion of the molecule. An added advantage to this approach is that the search library contains the MS of fragments instead of complete molecules. Because the size of the fragments is limited, it may actually be possible to construct most interesting molecules from a reasonably sized reference library of fragments [14].

NMR

Work in NMR has included both H^1 and C^{13} analysis. Proton NMR is similar to IR in that the protons in a functional group tend to absorb in certain regions of the spectrum regardless of the rest of the molecule. The range of possible absorptions for each functional group makes H^1 NMR more suitable for eliminating functional groups determined from another source than for generating a list of fragments [15]. The magnetic fields of the adjacent protons cause the peaks to split in predictable ways. It is possible to use these splitting patterns, called the multiplicity of the peaks, as a check of peak assignments and to test the final candidate structures.

C^{13} NMR, a more recent development, is very sensitive to the environment of the resonating carbon. This sensitivity causes every structurally unique carbon to resonate at a different frequency. The structural equivalence extends approximately two or three bonds in all directions. Structurally equivalent carbons in different molecules absorb at approximately the same frequencies. C^{13} NMR is therefore very good at generating a list of molecular fragments that are present. Work has been done to determine C^{13} NMR interpretation rules [16]; however, the general method used is a library search.

The library, in this case, is composed of molecular fragments and the characteristic absorption of the central carbon [17]. Unfortunately, it is a very time-consuming process to determine the specific carbon responsible for every peak in a spectrum. For this reason, the libraries used tend to be small. Because the characteristic absorptions overlap, a list of possible functional groups is generated for each NMR peak.

Internal Consistency Checks

Internal consistency checks eliminate fragments that are suggested by one source but can be shown to be impossible by another source. This elimination process helps reduce the combinatorial explosion of possible structures generated in the structure enumeration step. One of the most effective ways of providing this check is to combine information from different sources. Use of the molecular formula is the most straightforward example of this. For example, if there are no oxygens in the molecule, carbonyls, alcohols, and ethers can be eliminated. On the other hand, if there are two oxygens and the only oxygen-containing functional group is an alcohol, there must be two of them in the molecule. Consistency checks can also increase the confidence in the presence of a fragment if more than one source of data suggests its presence.

Combining information from multiple spectral sources is not as simple. It requires a detailed knowledge of the strengths and weaknesses of each type of spectroscopy in order to resolve conflicts between them. The problem is further complicated because most interpretation programs only return the probability that a functional group is present. This problem even goes beyond resolving the conflicts. In attempting to determine which of several possible functional groups is responsible for a given spectral feature, the confidence that one functional group is responsible can be affected by the confidence that other functional groups are present. Conflict resolution itself has been the subject of research [18].

In C^{13} NMR interpretation using library searching, consistency checking is an integral part of the process. For each peak present, a list of all possible structures responsible for the peak is retrieved from the library. The structure of the fragments for each peak is checked against that of the fragments for the other peaks. Fragments that would require the presence of a peak not present are eliminated. For example, if one possible explanation for a peak is a carbon with a methyl group and a carbonyl attached to it, there must be a peak somewhere in the spectrum that could have been caused by methyl carbons and carbonyl carbons. Otherwise, that fragment is not a reasonable explanation for the peak being identified.

Structure Enumeration

Once the various spectroscopic techniques have generated a list of molecular fragments, they must be connected to form possible structures for the unknown compound. The generation of possible structures must be both exhaustive and nonredundant. The simplest approach to the problem is to consider each fragment to be a "superatom," which can be connected to a specified number of other atoms.

Several problems arise during this procedure. The most serious problem is combinatorial explosion. Sometimes, there can be billions of possible structures. Enumeration programs must be able to eliminate chemically im-

possible structures and allow the chemist to eliminate other highly unlikely structures. Elimination is usually done at each step of a depth-first search to "prune" the search tree when possible. One advantage to a depth-first search is that all the structures below each branch point are eliminated when the branch point is eliminated. The other advantage is that the minimum number of assembled structures must be remembered at any given time.

Another problem in structure enumeration is that the list of fragments may contain ambiguities (i.e., the molecule contains either A or B but not A and B). Additionally, some input fragments may actually be part of another fragment in the input [19]. The programs must therefore be able to handle multiple possible starting points. If fragments are allowed to overlap or if the molecule contains more than one of any fragment, it is then possible to construct the same structure in multiple ways. The program should recognize when and where these redundancies occur and eliminate them.

The enumeration program must also recognize the existence of stereoisomers and distinguish between them [20-22]. This requirement limits the methods that can be used to represent the chemical structures (see Representing Chemical Structures Using a Computer).

Ranking of Candidate Structures

There are two methods of ranking candidate structures. The first method compares the unknown's spectrum with a predicted spectrum for the candidate structure. This requires the existence of a method to generate a spectrum from a structure. The DENDRAL project took this spectrum-comparing approach. The rules determining how a molecule will fragment were codified. Using these rules, the spectrum of a candidate structure could be predicted. These spectra are compared to the unknown's mass spectrum to determine the likelihood that the candidate structure is the correct one. Later work from the DENDRAL project automatically determined molecular fragmentation rules using the mass spectra of known compounds. This work was called meta-DENDRAL [23].

The same spectrum-comparing approach can be used for NMR spectra. There are two methods for predicting C^{13} NMR spectra. The first uses multiple linear regression to predict chemical shifts based on peak locations in similar compounds. The second approach uses a library search. This time the substructure of every carbon is searched, instead of the peaks. The predicted peak location is the average of all the peaks for that substructure in the library.

Proton NMR has the potential for comparing both the frequency and multiplicity of each peak. Correlation tables, analogous to those for IR, are used to assign NMR peaks to individual protons. Because of overlaps in the correlation tables, this is not always possible. If the peaks can be assigned, the multiplicity of each peak can be predicted and compared with the unknown's spectrum.

The second method for ranking structures discriminates against structures that are highly strained, such as three-membered rings. This technique is not as useful because, in general, not many of the possible structures are severely strained. In the end, if more than one possible structure remains, the chemist must devise tests to distinguish between the remaining candidates.

CHEMICAL REACTION SYNTHESIS

Chemical reaction synthesis is one of the oldest applications of AI in chemistry; it began in 1967 [24, 25]. This type of program attempts to design a sequence of chemical reactions that would result in a "target" molecule, and the early work was based on a chemical synthetic tree (Fig. 2). The target molecule was decomposed into its potential precursors using every possible single-step chemical synthesis. Each precursor was further decomposed into earlier antecedents, thus creating a synthetic tree. This decomposition led to a combinatorial explosion of potential synthetic paths. The deeper into a synthetic tree one proceeded, the more the number of potential paths multiplied. Initially, the selection of the best branch at each junction in the tree required the chemist's intervention in the program, which was necessary to limit the combinatorial explosion at each step. This simplification was accomplished using interactive computer graphics to display the potential paths, and the best precursor was selected by the chemist [26]. The program, "Simulation and Evaluation of Chemical Synthesis" (SECS), improved the interactive graphics by enabling the chemist to select the reaction path and broadened the computer knowledge base by adding stereochemical reactions and displays [3, 27].

One technique, which eliminated the need for a chemist's intervention to find a synthetic route, incorporated synthetic rules and heuristic programming [28]. The program uses only the molecular substructure of the target molecule that participate in the available synthetic methods. Heuristic rules determine the correct sequence of reactions required to protect any reactive functional groups on the target.

Expert systems techniques have also been applied to this problem. These methods, before further evaluation, reduce chemical reactions to more general axioms. SYNLM (synthesis with logic machine architecture) [29] uses theorem-proving techniques to design organic synthesis. Chemical reactions are converted into clauses, and these clauses are either true or false, depending on their arguments. This approach then uses a theorem prover to design organic syntheses. QED [30] applies multivalued logic predicate calculus with axioms. With this technique, the logic is not simply true or false but, rather, multivalued. Each rule has a confirmation value, a disconfirmation value, and a truth value. The values from all the applicable clauses are combined to predict "good" organic synthesis.

COMPUTER ALGEBRA APPLICATIONS

The fundamental theories in chemistry can be described by mathematical equations, which can be quite complex. Many chemical problems can be solved numerically using these equations [31, 32]. The solution of these equations has, however, been greatly simplified by the development of symbolic algebra packages [33]. These packages solve complex equations analytically, instead of using numerical approximations.

As early as 1954, symbolic algebra techniques were applied to problems in quantum chemistry [34]. However, only recently have symbolic algebra programs become popular [35]. This delay was caused by the high cost of computers powerful enough to run the software and the availability of commercial packages with full user support.

There are five commercially available packages that perform algebraic, calculus, and differential manipulations. They are MACSYMA [36,37], also available through EDUNET and ARPANET; REDUCE [38]; MAPLE [39]; SMP [40], written in C for speed of execution; and muMath [35,41], a micro-computer version. A useful feature of several of these packages is their ability to translate answers into FORTRAN code, which can be used by numerical programs.

Improving Chemical Instrumentation

IR Spectroscopy

Interpretation of IR spectra is one piece of the structure elucidation process described above. Many researchers have developed interpretation programs. One, the Program for the Analysis of IR Spectra (PAIRS), has been made widely available. PAIRS has been incorporated into several vendors' computerized Fourier transform IR spectrometers. This program gives the spectroscopist structural information for those spectra not found in the limited spectral library available on a spectrometer. PAIRS is a rule-based expert system written in FORTRAN. It uses the frequency, intensity, and width of peaks as input, as well as the molecular formula and solvent information. In addition to interpreting spectra, it can explain its reasoning to the spectroscopist. Recent work has also included developing better rules automatically using spectra of known compounds. The program is also being made available through QCPE [42].

Mass Spectroscopy

One of the most complex mass spectrometers today is the triple quadrupole mass spectrometer (TQMS). The TQMS is completely computerized, with more than 30 controllable parameters. Tuning the TQMS requires a high level of operator expertise because the controlled parameters interact with one another in complex ways. The interactions are too complex to be described by mathematical equations, but general rules of thumb have been developed. The tuning process must do more than maximize signal intensity. The shape of the resulting peak is also important. An expert system, TQMSTUNE, was developed to tune the instrument automatically [43,44]. The expert system is capable of outperforming a simplex optimization, which is a numerical scheme: however, it is still not quite as efficient as a competent operator.

X-Ray Powder Diffraction

Peak heights in x-ray powder diffraction of minerals are proportional to concentration, allowing quantitative analysis. Variations in relative peak heights from laboratory to laboratory and day to day preclude the use of simple least-squares fitting of reference spectra to the unknown. An expert system was developed to use the same knowledge that a mineralogist uses to solve this problem for sedimentary rocks [44,45].

The program must first determine which of about 25 minerals is present. This is complicated because each mineral has multiple peaks that can overlap or even coincide with peaks from other minerals. An iterative approach is used. The most likely major component is determined, and its spectrum is then subtracted from the sample spectrum. This process is repeated on

the resulting spectra until all that remains is noise.

This work was repeated using several different expert system development tools (EXPERT, UNITS, EMYCIN, OPS-5) and LISP. The study concluded that all the development tools and LISP had their shortcomings. The convenience of the expert system development tools can lead to restrictions that prevent the program from solving the problem completely. LISP, on the other hand, requires a great deal of programming effort. Fortunately, the knowledge base is significantly easier to translate from one system to another than it is to extract from the expert.

Chromatography

Expert Chromatography Assistance Team (ECAT) is an expert system designed to aid chemists in developing chromatography methods [44, 46]. It is a rule-based expert system using forward chaining to go from input problem parameters to the final separation method.

Methods development in chromatography is a task performed in most analytical laboratories, thus adding to the "payback" of the system. In designing a separation, the chemist requires certain chemical information concerning the analytes. That information is used to decide whether gas or liquid chromatography should be used. The information is also required to determine a column/mobile phase combination that can separate the components of interest. The detector chosen must be sensitive to all of the components. Many times, interferents or possible column contaminants must be removed before analysis can begin. The analyst determines both when this is necessary and how to perform the cleanup. Another common problem in chromatography is that peaks are not sufficiently resolved or require too much time to elute from the column. An optimization step is then required. Throughout this process it is assumed that the equipment is functioning properly. To make the expert system more reliable, the program should be able to determine whether unexpected results are caused by faulty equipment.

Currently, ECAT cannot perform all these tasks. However, because the tasks are separated into independent modules, the completed modules are useful while the others are still being developed. Currently, the chemical information data base, the column and mobile phase selection, and the column failure diagnosis modules are completed.

A PROLOG-base expert system has been developed to aid in methods development for separating steroids [47].

Ultracentrifuge

Ultracentrifugation is a technique for separating biological samples. As the separations that scientist's require become more difficult, ultracentrifugation has become increasingly complex. It is therefore increasingly difficult for most researchers to keep up with the technology. The expert system SpinPro questions the user on the nature of the sample and his/her research goals. This dialog is guided by the expert system, and the result is a recommended set of operating conditions designed to give optimum separation in minimum time [48]. SpinPro makes recommendations on all operating parameters, including rotor type, run speed, run time, temperature, tube materials, acceleration rates, gradient material, and gradient concentration.

SpinPro also recommends the best set of conditions using only equipment available in the researcher's lab. If this plan differs from the optimum,

Spinpro details how the compromises made influence the separation. The user interface will even make the ultracentrifugation expertise contained in the expert system available in a tutorial form. In this way, users can increase their own expertise. This expert system is a rule-based, reverse-chaining system that can be run on an IBM PC/XT.

Process Control

The monitoring and control of chemical process systems is a new area for expert systems. Control systems are directly connected to instruments that monitor the temperatures, pressures, concentrations, and other variables of the process equipment. These measurements are used to predict the development of the chemical process and, if problems are discovered, modify the controlling instrumentation to correct the process. These control systems are specific to each process and require extensive measurements of the chemical system and knowledge of process behavior when the controlling instrumentation is varied [49, 50].

THE FUTURE OF AI APPLICATIONS IN CHEMISTRY

Although AI offers the possibility of revolutionizing the practice of chemistry, there are many problems that must be overcome. One problem, which is easy to overlook until you develop your first expert system, is the time-intensive nature of extracting the knowledge from the expert and encoding it into an expert system. When you consider both the amount of chemical information available today and the rate at which it is being discovered, this problem may be the most severe. Fortunately, the whole of chemistry need not be encoded to develop useful systems. Even so, unless there are breakthroughs in knowledge engineering systems, the rate at which knowledge bases can be compiled will be a limiting factor.

A second major problem with AI application is the amount of computation required to solve a given problem. This problem is compounded because the most useful applications also tend to be the most complex. Several approaches to solving this problem are being investigated. Software approaches attempt to simply reduce the number of computations using either more efficient algorithms or better heuristics. Hardware solutions fall into two categories: faster processors that are designed specifically for AI and massively parallel architectures that work on more than one part of the problem at a time.

The driving force behind AI is so strong that not only will these problems be overcome but AI will become cost effective. As fast as the cost of computers has been dropping, the cost of doing chemistry has been skyrocketing. This has been caused by increases in chemists' salaries, costs of equipment, and costs of safely dealing with chemicals. AI can have an impact on all of these areas.

Expert systems can make chemists more productive. They can reduce the level of expertise required to solve a given problem and ease the drudgery of repetitive work, freeing up time for creative problem solving. They can also help minimize the loss of experience caused by retirements and attrition.

Expert systems can reduce instruments costs in several ways. If the instrument is more efficient at solving the problem, it will have increased throughput. The expertise to operate the instrument can be reduced. The

instruments can be designed to diagnose and correct their own problems, reducing downtime. Expert systems can be built to monitor plant processes to make them more efficient than a human operator.

Expert systems can also make the handling of chemicals safer. They can monitor equipment to notice problems before accidents occur and can make available the expertise to handle accidents should they occur.

One of the most exciting prospects of AI is to consider what possible problems it can solve. The time will come when the use of AI techniques will be an integral part of the practice of chemistry. AI will not replace people in the laboratory; rather, it will be a very useful assistant that handles details and allows the chemist to concentrate on more challenging problems.

REFERENCES

1. R. K. Lindsay, B. C. Buchanan, E. A. Feigenbaum, and J. Lederberg, *Application of Artificial Intelligence for Organic Chemistry: The DENDRAL Project*, McCraw-Hill, New York, 1980.
2. R. K. Lindsay, B. C. Buchanan, E. A. Feigenbaum, and J. Lederberg, *Applications of Artificial Intelligence for Chemical Inference: The DENDRAL Project*, McCraw-Hill, New York, 1980.
3. W. T. Wipke, S. R. Heller, R. J. Feldman, and E. Hyde, ed., *Computer Representation and Manipulation of Chemical Information*, John Wiley, New York, 1974, pp. 147-174.
4. H. W. Whitlock, "An Organic Chemist's View of Formal Language," *Computer-Assisted Organic Synthesis* (T. W. Wipke, and J. Howe, eds.) American Chemical Society Symposium Series 61, Washington, D.C., 1977.
5. W. J. Wiswesser, *A Line-Formula Chemical Notation*, Thomas Y. Crowell, New York, 1954.
6. E. C. Smith and P. A. Baker, *The Wiswesser Line-Formula Chemical Notation*, 3rd ed, CIMI, NJ, 1975.
7. S. H. Bertz, W. C. Herndon, and C. Dabbagh, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 169-175.
8. J. E. Gordon and J. C. Brockwell, "Chemical Inference," *J. Chem. Inf. Comp. Sci.*, 23, 117 (1983).
9. Z. Hippe, "Problems in the Application of AI in Analytical Chemistry," *Anal. Chim. Acta*, 150, 11-21 (1983).
10. N. B. Colthup, *J. Opt. Soc. Am.*, 40, 397-400 (1950).
11. B. Schrade et al., "Automatic Reduction and Evaluation of IR and Raman Spectra," *F. Z. Anal. Chem.*, 303, 337-348 (1980).
12. H. B. Woodruff and M. E. Munk, "Computer-Assisted Interpretation of IR Spectra," *Anal. Chim. Acta*, 95, 13-23 (1977).
13. H. B. Woodruff, S. A. Tomellini, and G. M. Smith, "Elucidation of Structural Fragments by Computer-Assisted Interpretation of IR Spectra," *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington D.C., 1986, pp. 312-320.
14. K. P. Cross, A. B. Giordani, H. R. Cregg, P. A. Hoggman, C. F. Beckner, and C. G. Enke, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 321-336.

15. H. Egli, D. H. Smith, and C. Djerassi, "Computer-Assisted Structural Interpretation of Proton NMR Spectral Data," *Hel. Chim. Acta*, **65**, 1898-1919 (1982).
16. T. M. Mitchell and G. M. Schwenzer, "Application of Artificial Intelligence for Chemical Inference. XXV. A Computer Program for Automated Empirical C¹³ NMR Rule Formation," *Org. Magn. Reson.*, **11**(8), 378-384 (1978).
17. M. R. Lindley, N. A. B. Gray, D. H. Smith, and C. Djerassi, "Applications of AI for Chemical Inference 40. Computerized Approach to Verification of C¹³ NMR Spectral Assignments," *J. Org. Chem.*, **47**, 1027-1035 (1982).
18. B. Curry and J. A. Michnowicz, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 350-364.
19. R.E. Carhart, D. H. Smith, N. A. B. Gray, J. G. Nourse, and C. Djerassi, "GENOA: A Computer Program for Structure Elucidation Utilizing Overlapping and Alternative Substructures," *J. Org. Chem.*, **46**, 1708-1718 (1981).
20. J. G. Nourse, R. E. Carhart, D. H. Smith, and C. Djerassi, "Exhaustive Generation of Stereoisomers for Structure Elucidation," *J. Am. Chem. Soc.*, **101**, 1216-1223 (1979).
21. J. G. Nourse, "The Configuration Symmetry Group and Its Application to Stereoisomer Generation, Specification, and Enumeration," *J. Am. Chem. Soc.*, **101**(5), 1210 (1979).
22. J. G. Nourse, D. H. Smith, R. E. Carhart, and C. Djerassi, "Computer-Assisted Elucidation of Molecular Structure with Stereochemistry," *J. Am. Chem. Soc.*, **102**, 6289-6295 (1980).
23. B. G. Buchanan, D. H. Smith, W. C. White, R. J. Gritter, E. A. Feigenbaum, J. Lederberg, and C. Djerassi, "Applications of Artificial Intelligence for Chemical Inference. 22. Automatic Rule Formation in Mass Spectrometry by Means of the Meta-DENDRAL Program," *J. Am. Chem. Soc.*, **98**(20), 6168-6178 (1976).
24. E. J. Corey, "General Methods for the Construction of Complex Molecules," *Pure and Appl. Chem.*, **14**, 19 (1967).
25. E. J. Corey and T. W. Wipke, "Computer-Assisted Design of Complex Organic Synthesis," *Science*, **166**(178) (1969).
26. E. J. Corey, "OCCS-Organic Chemical Simulation of Synthesis, LHASA-Logic and Heuristics Applied to Synthetic Analysis," *J. Am. Chem. Soc.* (1972).
27. T. Wipke and T. Dyott, "Simulation and Evaluation of Chemical Synthesis," *J. Am. Chem. Soc.*, **96**(15), 4825 (1974).
28. P. E. Blower, Jr., and H. W. Whitlock, Jr., "An Application of Artificial Intelligence to Organic Synthesis," *J. Am. Chem. Soc.*, **98**(6), 1499-1510 (1976).
29. T. Wang, I. Burnstein, S. Ehrlich, M. Evens, A. Gough, and P. Johnson, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 244-257.
30. W. T. Wipke, and D. P. Dolata, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 188-208.
31. Quantum Chemistry Program Exchange, Indiana University, Bloomington, IN.
32. S. A. Borman, "Scientific Software," *Anal. Chem.*, **57**(9), 983A (1985).

33. J. F. Ogilvie, "Applications of Computer Algebra in Physical Chemistry," *Comp. Chem.*, **6**, 169-172 (1982).
34. S. E. Boys, B. G. Cook, C. M. Reeves, and I. Shavitt, *Nature* **178**, 1207-1209 (1956).
35. C. S., Johnson, "Computer Algebra in Chemistry," *J. Chem. Inf. Comp. Sci.*, **23**, 151-157 (1983).
36. *MACSYMA Reference Manual*, MIT Mathlab Group, Cambridge, MA, 1977.
37. *MACSYMA Primer*, MIT Mathlab Group, Cambridge, MA, 1982.
38. A. C. Hearn, ed., *REDUCE User's Manual*, Version 3.0, Rand Publication CP78(4/83), The Rand Corp., Santa Monica, CA, 1983.
39. K. O. Geddes, G. H. Gonnet, and B. W. Char, *MAPLE User's Manual*, 2nd. ed., University of Waterloo, Waterloo, Ontario, Canada N2L3G1.
40. C. A. Cole, S. Wolfram, et al. *SMP Handbook*, Caltech, Pasadena, CA, 1981.
41. G. Williams, *BYTE*, **5**, 325-338 (1980).
42. Quantum Chemistry Program Exchange, Indiana University, Bloomington, IN, Program 497.
43. C. Wong, and S. Lanning, "AI in Chemical Analysis," *Energy and Technology Review*, Lawrence Livermore National Laboratory, February 1984.
44. R. E. Dessy, ed., "Expert Systems Part II," *Anal. Chem.*, **56**(12), 1312A-1332A (1984).
45. S. P. Ennis, "Expert Systems, A User's Perspective of Some Current Tools," *Proceedings of the National Conference on AI*, William Kaufmann, Los Altos, CA, 1982, pp. 319-321.
46. J. Karnicky, R. Bach, and S. Abbott, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 278-296.
47. H. Gunasingham, B. Srinivasan, and A. L. Ananda, "Design of a PROLOG-Based Expert System for Planning Separation of Steroids by HPLC," *Anal. Chim. Acta*, **182**, 193-203 (1986).
48. P. R. Martz and M. Heffron, *Artificial Intelligence Applications in Chemistry*, American Chemical Society, Symposium Series 306, Washington, D.C., 1986, pp. 297-311.
49. R. L. Moore, C. G. Knickerbocker, and L. B. Hawkinson, *Artificial Intelligence Applications in Chemistry*, American Chemical Symposium Series 306, Washington, D.C., 1986, pp. 69-74.
50. E. A. Scarl, J. R. Jamieson, and C. I. DeLaune, "Process Monitoring and Fault Location at the Kennedy Space Center," *SIGART Newsl.*, **93**, 38 (1985).
51. R. Cornelius, D. Cabrol, and C. Cachet, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 125-135.
52. B. Hohne and R. Houghton, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 87-99.
53. J. C. Bellows, "Artificial Intelligence Chemistry Diagnostic System," *Proceedings of the 45th International Water Conference*, Engineering Society of Western Pennsylvania, 1984, pp. 15-25.
54. J. C. Bellows, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, 52-68.
55. D. Garfinkel, L. Garfinkel, V. W. Soo, C. A. Kulikowski, *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 75-86.

56. C. E. Reise and J. D. Stuart, "A Knowledge-Engineering Facility for Building Scientific Expert Systems," *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 18-30.
57. H. L. Keith and J. D. Stuart, "A Rule-Induction Program for Quality Assurance-Quality Control and Selection of Protective Materials," *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 31-52.
58. A. L. Smith, "A Rule-Based Declarative Language for Scientific Equation Solving," *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 111-118.
59. D. Edelson, "A Chemical-Reaction Interpreter for Simulation of Complex Kinetics," *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 119-124.
60. R. Banares-Alcantra, A. W. Westerberg, and M. D. Rychener, "Development of an Expert System for Physical Property Predictions," *Comput. Chem. Eng.*, 9(2), 127-142 (1985).
61. T. J. Harner, G. C. Levy, E. J. Dudewicz, F. Delaglio and A. Kumar, "Artificial Intelligence, Logic Programming, and Statistics in Magnetic Resonance Imaging and Spectroscopic Analysis," *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 337-349.
62. S. A. Liebman, P. J. Duff, M. A. Schroeder, R. A. Fifer, and A. M. Harper, "Concerted Organic Analysis of Materials and Expert System Development," *Artificial Intelligence Applications in Chemistry*, American Chemical Society Symposium Series 306, Washington, D.C., 1986, pp. 365-385.
63. I. Fujiwara, T. Okuyama, T. Yamasaki, H. Abe, and S. Sasaki, "Computer-Aided Structure Elucidation of Organic Compounds with the CHEMICS System," *Anal. Chim. Acta*, 133, 527-533 (1981).
64. S. Sasaki, H. Abe, I. Fujiwara, and T. Yamasaki, "The Application of C^{13} NMR in CHEMICS, the Computer Program System for Structure Elucidation," *Stud. Theor. Chem.* 16, 186-204 (1981).
65. M. E. Munk, C. A. Shelley, H. B. Woodruff, and M. O. Trulson, "Computer-Assisted Structure Elucidation," *F. Z. Anal. Chem.*, 313, 473-479 (1982).
66. C. A. Shelley and M. E. Munk, "CASE, A Computer Model of the Structure Elucidation Process," *Anal. Chim. Acta*, 133, 507-516 (1981).
67. S. Sasaki, H. Abe, I. Fujiwara, T. Yamasaki, Z. Hippe, B. Debska, J. Duliban, and B. Guzowska-Swider, "Recent Problems of Application of Artificial Intelligence in Computer-Aided Elucidation of Chemical Structures," *Chem. Anal. (Warsaw)*, 27(3-4), 171-181 (1982).
68. B. Debska, J. Duliban, B. Guzowska-Swider, and Z. Hippe, "Computer-Aided Structural Analysis of Organic Compounds by AI System," *Anal. Chim. Acta*, 133, 303-318 (1981).
69. L. A. Gribov, M. E. Elyashberg, and V. V. Serov, "Computer System for Structure Recognition of Polyatomic Molecules by IR, NMR, UV, and MS Methods," *Anal. Chim. Acta*, 95, 75-96 (1977).
70. "EXPERTISE Uses IR Spectra for Chemical Evaluation," *Intell. Instrum. Comput.*, 4(4-S), 1-4 (1986).

71. P. Harmon, ed., "Inventory and Analysis of Existing Expert Systems," *Expert Syst. Strategies*, 2(8), 1-17 (1986).
72. K. Janssens and P. Van Espen, "Implementation of an Expert System for the Qualitative Interpretation of X-Ray Fluorescence Spectra," *Anal. Chim. Acta*, 184, 117-132 (1986).

APPENDIX A: GLOSSARY

Artificial Intelligence (AI): A field of computer science that attempts to use computers to perform tasks viewed as requiring "intelligence." In chemistry applications, AI systems may assist in identifying compounds and suggesting formulations.

Backward Chaining: A technique in logical deduction using IF-THEN rules, whereby a conclusion (THEN clause) is proved by searching backwards to prove all the hypotheses (IF clauses). Each IF clause can be a THEN clause in a different rule; therefore, this technique "chains backwards" through an arbitrary number of levels in an attempt to prove the initial premise. For example, the reactants in a reaction formula may be the product in another reaction formula, thereby allowing the scientist to trace the reaction back to simple starting materials.

Certainty Factor (CF): A weighted value in a range of numbers (e.g., 0 to 1 or -1 to 1) that denotes the confidence in the validity of a conclusion. CFs are used in expert systems to measure the value of the conclusion (THEN clause). Chemists may assign a certainty value to an answer to measure how confident they are that a conclusion is valid.

Combinatorial Explosion: A mathematical property that causes a large number of solution paths to be generated in attempting to solve a problem. AI systems incorporating heuristic reasoning limit the number of solution paths that they must evaluate.

Domain: The area of study covered by an expert system; in this case, a field of chemistry.

Domain Expert: The person whose expertise in the domain is the basis of the expert system. The knowledge base of an expert system is formed by including the decision-making rules used by the domain expert.

Expert System: A computer program designed to solve problems in a specialty area. Expert systems consist of a knowledge base, an inference engine, and a user interface.

Explanation Facility: The part of an expert system's user interface that explains how the program reached a particular conclusion. This facility acts as a logic-tracking system for the user and is often a key reason for an end user accepting an expert system.

Forward Chaining: A technique in logical deduction using IF-THEN rules, where all possible conclusions are drawn from known facts. Because each new conclusion (THEN clause) can be used as a fact (IF clause) to derive other conclusions, the program "chains forward" until it has used all relevant rules in the knowledge system. In formulations chemistry, forward chaining is used when starting materials are known and a chemist wants to find an optimal compound formulation.

Frames: A hierarchical method of representing rules in the knowledge base of an expert system. In using a hierarchy, each rule associated with a

fact is either stated explicitly or inherited from a higher level. Expert systems using frames will search hierarchies to associate facts with conclusions.

Graph Theory: An area of mathematics that establishes relationships between objects by representing their connectedness using graphs. In chemical AI systems, this theory is applied as a method of representing chemical structures and properties by connected graphs.

Heuristic: An expert's rule of thumb used in solving a problem. Heuristics are shortcuts that can speed up the time a program takes to solve a problem, but using heuristics does not guarantee that an existing solution will be found. This limitation is acceptable in cases where other methods yield a great number of possible solutions that must be explored.

Inference Engine: The part of an expert system that performs the logic calculations using the knowledge base and user's input. The engine determines which rules are appropriate and under which conditions to use the rules. AI systems may use heuristic methods to arrive at a solution faster.

Intelligent Assistants: Expert systems designed to solve parts of a problem. These smaller systems assist an expert in solving an entire problem.

Knowledge Base: The part of an expert system that contains the information about the problem area. Different methods for representing this information are used by expert system programmers. Researchers are continuing to look for an optimal method of representing information in a knowledge base.

Knowledge Engineer: The person responsible for representing a domain expert's knowledge in a knowledge base. This person writes rules in a knowledge base, which reflect the way an expert reaches decisions.

LISP: A programming language used to write AI systems. LISP is an acronym for list processor.

Multilevel Expert System: An expert system that takes advantage of traditional numerical calculations in its decision-making process. In a chemical expert system, conclusions may be dependent on results of calculations on molecules, as well as on a molecule's characteristics.

Natural Language Processing: A method of translating English sentences into statements that computer programs understand (and vice versa). In chemistry applications, an AI system using natural language processing will read and interpret chemical structures represented in notations that a chemist understands.

Predicate Calculus: Logical deductions based on the descriptive attributes of an object. In chemical AI applications, conclusions may be drawn about a formulation based on the attributes of a molecular compound.

Production Rule: Another name for IF-THEN rules used in representing information in an expert system's knowledge base.

Production System: Another name for an expert system.

Prolog: A programming language commonly used to code AI application programs. PROLOG is an acronym for programming in logic and is used prevalently in Europe.

Symbolic Processing: A method used in AI applications that allows computer programs to manipulate non-numerical data. This processing technique uses symbols to represent concepts, or pieces of information, instead of numerical data.

APPENDIX B: EXPERT SYSTEMS FOR CHEMISTRY

Acronym	Reference	Language	Application
GEORGE	51	?	Chemistry tutor for problems involving mass, volume, and moles.
None	52	LISP	Formulates commercial agricultural products from biologically active chemicals.
None	53, 54		Diagnoses water chemistry problems in steam power plants. Recommends corrections and the results of inaction.
None	55	EXPERT	Intelligent program interface helps perform complex equilibrium calculations, determines kinetic models, and designs drug studies
TOGA	56	Rulemaster	Fault diagnosis of transformers using chemical analysis of the transformer oil.
GloveAid	57	Rulemaster	Determines hand protection for specific hazards.
QualAid	57	Rulemaster	Advice on quality assurance and quality control for environmental analysis.
MACSYMA	36, 37	LISP	Computerized algebra and calculus.
REDUCE	38	LISP	Computerized algebra and calculus.
MAPLE	39	LISP	Computerized algebra and calculus.
SMP	40	C	Computerized algebra and calculus.
muMath	35, 41	LISP	Computerized algebra and calculus.
TK! Solver	58	?	Solves sets of simultaneous equations.
None	59	FORTTRAN	Interprets chemical kinetics equations to generate differential equations.
TQMSTUNE	43, 44	KEE	Tunes a triple quadrupole mass spectrometer (MS/MS).

(continued)

APPENDIX B (continued)

Acronym	Reference	Language	Application
PAIRS	42	FORTTRAN	Interprets IR spectra.
DENDRAL	1, 2	LISP	Interprets MS and NMR spectra and generates molecular structures from fragments of the molecule.
ECAT	44	LISP	Aids in developing separation methods using high-performance liquid chromatography.
None	44	LISP	Quantitative analysis of x-ray diffraction spectra.
CONPHYDE	60	PROSPECTOR	Determines which vapor-liquid equilibrium calculation program is appropriate.
SpinPro	48	LISP	Determines optimum equipment configurations for ultracentrifuge separations.
GENOA	19	LISP	Exhaustive generation of molecular structures from fragments; part of the DENDRAL project.
MRI, LOG, ESP	61	PROLOG	Analysis of NMR and magnetic resonance imaging data.
None	18	?	Organic structure elucidation combining IR and MS data.
EXMAT	62	TIMM	Development of chemical analysis methods using multiple types of instruments.
CHEMICS	63, 64	FORTTRAN	Organic structure elucidation using NMR, IR, and MS data.
CASE	65, 66	FORTTRAN	Organic structure elucidation using NMR and IR data.
SEAC	67, 68		Organic structure elucidation using NMR and IR data.
STREC	69	FORTTRAN	Organic structure elucidation using NMR, IR, and MS data.
None	47	PROLOG	Methods development for HPLC separations of steroids.
EXPERTISE	70	?	Interpretation of IR spectra.
Chemical Plant Control	71	PICON	Proprietary expert system at Exxon for chemical plant control optimization.

(continued)

APPENDIX B (continued)

Acronym	Reference	Language	Application
Chemical Process Simulator	71	PICON	Proprietary expert system at Eastman Kodak for simulating complex chemical processes.
Herbicide Advisor	71	?	Proprietary expert system at Shell for identifying new herbicides based on chemical structure.
None	71	PICON	Commercial product at Texaco for optimizing process control.
X-ray Diffractometer Advisor	72	Personal Consultant	Proprietary expert system at Westinghouse for identifying crystallographic structures of metals.
None	73	Pascal	Qualitative interpretation of energy-dispersive x-ray spectra.

BRUCE A. HOHNE
THOMAS H. PIERCE
MARY A. BRIGHT

ASCII CODE

INTRODUCTION

When the UNIVAC 1 was first developed, the market analysts of the day forecast the need for a maximum of 6 to 10 computers worldwide. With thinking such as this, there was clearly no pressing need to develop any standards as to how these machines might communicate.

However, as the transistor made its impact, computers began to appear by the hundreds in the late 1950s. Each manufacturer was free to use his own code for the representation of the letters, numbers, and other symbols necessary for the processing of data.

Complicating the problem was the fact that the technology of the time limited the number of combinations available. For instance, a printer might have room for only 48 or 64 different characters. Different installations had different needs and, frequently, different character sets. Thus, a certain code combination might convert to the symbol # in a business data processing computer, but it would be the symbol = in a scientific environment. This was particularly troublesome to businesses owning more than one computer, as the transfer of data from one to another involved unproductive and, frequently, complicated conversions for it to be useful.

HISTORICAL DEVELOPMENT OF THE ASCII CODE

A number of user groups saw the problem of nonuniformity developing as early as the late 1950s and attempted to do something about it. However, none of them was powerful enough to establish a de facto standard.

This type of problem was not new. In the early part of the century, the need for standards in newly evolving technical areas became widespread. This need produced the formation of organizations for that particular purpose. On the international scene, the International Organization for Standardization (ISO) was formed and, domestically, the American National Standards Institute (ANSI) [1].

The predecessor organization to ANSI, the American Standards Association (ASA), became the logical place where a resolution to the code problem was to be found. Early deliberations by ASA established that none of the existing codes was suitable as a standard, because they neither met the requirements of the time nor offered provisions for future requirements. Eventually, a design for a 7-bit code emerged. Six-bit codes are limited to 64 combinations, and as we saw earlier, this was not enough to uniquely provide for the comprehensive needs of all users.

In 1963, the first standard was adopted by ASA, the U.S.A. Standard Code for Information Interchange (USASCII), which was subsequently shortened to ASCII (pronounced "asskee"). The original standard assigned only 100 of the 128 possible combinations. Later revisions would use all of those positions and assign 26 of them for lowercase characters.

The standard was evolved primarily to facilitate the external exchange of information between machines, regardless of the internal codes used. However, it became apparent that there were advantages to its use internally as well, such as the elimination of the need for code conversions on input and output.

Internal use of the ASCII code had many advantages. Bemser [2], who served as an alternate member on the standard-making committee, cited several, which follow:

1. Manipulation by classes. Because all characters of certain classes are grouped contiguously, they can be classified with few instructions.
2. Fewer instruction scans because of unique codes. The alternative is to determine a code's meaning in syntax because it could be used with multiple translations.
3. Faster and cheaper sorting when the collating sequence is identical to the binary sequence of the codes for graphics.
4. Reduction in the number of routines required to be programmed.
5. Fewer tables for mixed codes in communications.
6. Clarity of printed output, particularly in the reproduction of the source program.
7. A tendency for keyboards to be identical with typing communications equipment.

Since 1963 the code has undergone an evolution and has come into wide use throughout the United States and Canada. The original standard has been revised several times.

USA Standard X3.4-1967, approved March 11, 1968

USA Standard X3.4-1968, approved October 10, 1968

ANSI Standard X3.4-1977, approved June 9, 1977 [3]

Initially, use of a 7-bit code did present some problem to machines of the 1963 era because nearly all of them were designed using a 6-bit character code. As a result, the new code did not really become established until well after IBM introduced the 8-bit byte in its 360 series in late 1964.

Because IBM embraced its own 8-bit Extended Binary Coded Decimal Interchange Code (EBCDIC), the standard was not universally accepted even then. However, most other manufacturers adopted the ASCII standard and it has gradually established itself as the preferred standard today. Even IBM adopted it for internal use in its personal computer in 1981.

Related standards have also been developed to facilitate extensions to the original ASCII code and implementation on various physical devices. These have become necessary because of the development of technology, which provides for increasingly sophisticated functions that must be provided for in any standard code.

A revised X3.4 standard was circulated during 1985, and it is expected to be approved soon. There are also other related standards in process:

X3.134.1, Eight-Bit ASCII Structure and Rules

X3.134.2, Seven-Bit and Eight-Bit ASCII

For more information, contact the American National Standards Institute, 1430 Broadway, New York, New York 10018.

DESIGN CONSIDERATIONS (X3.4)

The 3.4 standard was developed in parallel with its international counterpart, ISO 646-1973. Final acceptance of the code was attained chiefly because the membership of X3.4 design subcommittee represented virtually all of the diverse interests involved in the use of such codes.

The primary areas to be standardized were set size, set structure, character selection, and character placement of the code. Specifically, these areas included the need for an adequate number of graphic symbols, the need for an adequate number of control codes and information separators, the desire for an unambiguous code in which each code has a unique meaning, physical requirements of media and facilities, error control considerations, special interpretation of the all-zeros and all-ones characters, ease in identification of classes of characters, data manipulation requirements, collating conventions, keyboard conventions, other set sizes, international considerations, programming languages, and existing coded character sets.

SET SIZE

At present, 7 bits is the minimum size that will meet the requirements for graphics and controls in applications involving information interchange. As was stated earlier, there is also the X3.41-1974 standard, which provides a procedure for 8-bit extensions, and a proposed 8-bit standard was circulated in 1985 and is expected to be approved [4].

SET STRUCTURE

To describe the set structure, the standards subcommittee decided it was most convenient to use a format of 8 columns and 16 rows (see Fig. 1).

The code was also designed to represent graphics characters (letters, numbers, and special printable symbols) in a dense (no unused codes), contiguous sequence of codes. Thus, the graphics characters are represented in Figure 1 in columns two through seven.

Control characters complete the table in columns zero and one. There were several reasons for this. First, the null character (all zero bits) by definition has to be the first character, and it is generally perceived as a control character. Further, the codes represented by column seven were felt to be most susceptible to certain types of transmission errors, and it would be best to avoid their use for important control codes. A third reason was the fact that because of considerations of the graphic subset structure, the two control columns had to be adjacent. The resulting code structure enables the easy identification of classes of graphics and controls.

CHOICE OF GRAPHICS

The graphics set includes all upper- and lowercase alphabetic characters, as well as the digits 0 through 9. It also includes those punctuation, mathe-

Bits					COLUMNS									
b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	0	1	2	3	4	5	6	7
ROWS							0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	NUL	DLE	SP	0	@	P	\	p
0	0	0	1	1	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	2	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	3	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	4	4	EOT	DC4	\$	4	D	T	d	t
0	1	0	1	5	5	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	6	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	7	7	BEL	ETB	/	7	G	W	g	w
1	0	0	0	8	8	8	BS	CAN	(8	H	X	h	x
1	0	0	1	9	9	9	HT	EM)	9	I	Y	i	y
1	0	1	0	10	10	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	11	11	VT	ESC	+	;	K	[k	{
1	1	0	0	12	12	12	FF	FS	,	<	L	\	l	
1	1	0	1	13	13	13	CR	GS	-	=	M]	m	}
1	1	1	0	14	14	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	15	15	SI	US	/	?	O	_	o	DEL

FIGURE 1 The American National Standard Code for Information Interchange (ASCII). This table is arranged to allow the character positions to be specified in column/row character notation, where the column number is the decimal equivalent of bits b₇ through b₅ in a 7-bit environment (or bits a₈ through a₅ in an 8-bit environment), and the row number is the decimal equivalent of bits b₄ through b₁ or bits a₄ through a₁, with b₇ (or a₈) the most significant bit in the character. Bit combinations are represented by the notation "column m, row n" or alternatively as "m/n", where m and n are the decimal and row numbers, respectively. For instance, the capital letter A is the character in position 4/1.

matical, and business symbols considered most useful. Among them are the characters commonly encountered in programming languages such as COBOL and FORTRAN.

To enable representation of foreign languages, two diacritical (or accent) marks have been included, and provision has been made for the use of four punctuation symbols alternatively as diacritical marks in conjunction with a back space.

GRAPHIC SUBSET STRUCTURE

The basic structure of the graphic subset was influenced by several things: the logical collating considerations, the relationship to related 6-bit sets, and the needs of typewriterlike devices.

To minimize processing time, it was desirable for the logical ordering of characters to be accomplished by simple comparison of binary codes. It also required that the space character sort before any other symbol. The

requirement that punctuation symbols collate ahead of the alphabet established their position in the code as well.

To simplify the design of typewriterlike devices, it was desirable that there be only a 1-bit difference between characters paired on keys. This plus the requirements for continuity and the collating sequence resulted in the placement of the graphic subset in the last four columns.

It was expected that some devices would have limited graphic reproduction capability, such as a 64-character set. Thus, a single bit difference between upper- and lowercase, combined with the contiguous requirement, was necessary to allow easy translation of lowercase codes to uppercase when used on such devices.

To minimize ambiguity caused by the use of a 64-character (6-bit) device, each character in columns six and seven should differ little in significance from the corresponding character in columns four and five. In some cases, this was not possible.

The resultant collating sequence of "specials" (S), "digits" (D), and "alphabetics" (A) does not conform to the most prevalent convention (S-A-D) in common usage and in other codes, such as EBCDIC, because of other code requirements.

The need for a simple transformation from the ASCII collating sequence to the more prevalent convention was recognized. This dictated placement of some of the specials within the set. Specifically, those symbols that are most often used as identifiers for ordering information and normally collate before letters and digits (i.e., ampersand [&], comma [,], asterisk [*], etc.), were not placed in a column containing digits. Thus, the entire numeric column could be rotated to a position higher than the alphabet by a simple transformation.

The need for a mathematically oriented 4-bit subset also played a role in the placement of characters. Such a 4-bit (16 possibilities) subset, including the digits and the symbols asterisk, plus, comma, hyphen, period, and slant can be derived easily from the code.

Provision was made for the accommodation of alphabets containing more than 26 letters by the location of low-usage characters in the area following the alphabet.

CONTROL SUBSET AND STRUCTURE

The control characters included in the set are those required for control of terminal devices, input and output devices, format, or communication transmission and switching. They fall into the following categories:

- Communication controls
- Format effectors
- Device controls
- Information separators

Information separators are intended to identify machine sensible boundaries of various data elements. They consist of file separator (FS), group separator (GS), record separator (RS), and unit separator (US). The exact nature of their use within data is not specified.

The synchronous idle (SYN) character was located so that its binary pattern in serial transmission was unambiguous as to character framing. The

acknowledge (ACK)/negative acknowledge (NAK) characters were located so as to gain the maximum protection against mutation of one into the other through transmission errors.

Many output devices require a carriage return (CR) in combination with a line feed (LF) to move to the first position of the next line. ASCII associates a function, "new line" (NL) with the LF character for devices that do not require both symbols. Thus, the sequence CR-LF could produce the same result on devices of both classes.

EXTENSION TECHNIQUES (X3.41)

In 1974 ANSI Standard 3.41 was approved, which provided techniques for extending the code in both a 7- and an 8-bit environment [4]. More exacting standards are now under consideration.

The most common method used in the microcomputer world for extension of the ASCII code is known as the "escape sequence." This is a bit string consisting of two or more bit combinations, the first of which is the combination corresponding to the "escape" character.

Frequently, microcomputer printers will be controlled by sending escape sequences to control device functions, such as performing a form feed, changing type size or pitch, and so forth.

RELATED STANDARDS

The X3.4 standard applies only to the configuration of the code itself. Related standards have been promulgated as to how the code is to be implemented on specific physical devices. Some of these include

- X3.6 Perforated Tape Code for Information Interchange
- X3.14 Recorded Magnetic Tape for Information Interchange (200 cpi, NRZI)
- X3.15 Bit Sequencing for ASCII in Serial by Bit Data Transmission
- X3.16 Character Structure and Character Parity Sense for Serial-by-Bit Data Communication in ASCII
- X3.17 Character Set and Print Quality for OCR (OCR-A)
- X3.22 Recorded Magnetic Tape for Information Interchange (800 cpi, NRZI)
- X3.25 Character Structure and Character Parity Sense for Parallel-by-Bit Data Communication in ASCII
- X3.26 Hollerith Punched Card Code
- X3.27 Magnetic Tape Labels and File Structure for Information Interchange
- X3.28 Procedures for Use of Communication Control Characters of ASCII in Data Communication Links
- X3.32 Graphic Representation of the Control Characters of ASCII
- X3.39 Recorded Magnetic Tape for Information Interchange (1600 cpi, PE)
- X3.41 Code Extension Techniques for Use with 7-bit ASCII
- X3.45 Specifications for Character Set Handprinting

- X3.48 Magnetic Tape Cassettes for Information Interchange
- X3.49 Character Set for OCR (OCR-B)
- X3.54 Recorded Magnetic Tape for Information Interchange (6250 cpi, Group-coded recording)
- X3.56 Recorded Magnetic Tape Cartridge for Information Interchange
- X3.57 Structure for Formulating Message Headings for Information Interchange using ASCII
- X3.64 Additional Device Controls for Use with ASCII
- X4.14 Alphanumeric Keyboard Arrangements Accommodating the ASCII and American National Standard Character Set for Optical Character Recognition

REFERENCES

1. C. Card, R. D. Prigge, J. L. Walkoxicz, and M. F. Hill, "The World of Standards," *Byte*, 130-142 (February 1983).
2. R. W. Bemer, "The American Standard Code for Information Interchange, Part Two," *Datamation*, 9, 39-44 (September 1963).
3. *ANSI X3.4-1977: American National Standard Code for Information Interchange*, American National Standards Institute, New York, 1977.
4. *ANSI X3.41-1974: Code Extension Techniques for Use with the 7-bit Coded Character Set of American Standard Code for Information Interchange*, American National Standards Institute, New York, 1974.

LEE C. TADDONIO

ASTHON—TATE, INC.

OVERVIEW

Ashton-Tate, a worldwide leader in the development and marketing of microcomputer business software, is the number one supplier of data base management systems. The company also offers advanced integrated and word processing packages and provides a level of customer service and support that has become the industry standard.

Based in Torrance, California, Ashton-Tate was founded in 1980 and has since become one of the fastest growing companies in the personal computer industry. Revenues of \$465,000 for the company's first 5 months of operation have grown to more than \$210.8 million in 1987. Ashton-Tate was the first microcomputer software company to bring real data base management to the personal computer and now provides software packages that increase business productivity significantly. Among its current products, dBASE III PLUS is the most complete data base management system for microcomputer users, and the integrated package, Framework II, combines the power of a spreadsheet and word processor. Ashton-Tate also produces dBASE II, the first full-function relational data base management system.

The company began as a mail-order software house in 1980 when cofounders George Tate and Hal Lashlee took a data base management program written by a young engineer named Wayne Ratliff (Ratliff would later join Ashton-Tate as the company's chief scientist) and began marketing it as the software solution for managing large amounts of information. dBASE II, Ashton-Tate's first product, assembled, sorted, and organized large quantities of data with a programming language that could be customized for individual business needs.

Success came quickly for the young company. Not only did Tate and Lashlee have an excellent product, they approached the software industry with a unique, full-service marketing strategy. At a time when other software companies focused all of their energies on developing sophisticated software, the two entrepreneurs pursued an additional path—offering quick delivery, volume discounts, and full support services, including a toll-free telephone number.

With his company expanding to more than 250 employees between 1980 and 1983, Tate began to form a team of experienced managers who could effectively guide the company through this crucial growth period. In mid-1982, David Cole took over the day-to-day operation of the company as Ashton-Tate's president. During Cole's tenure, the company introduced a more sophisticated data base management product called dBASE III. Cole also added Framework to Ashton-Tate's product line. Developed by Forefront Corporation of Sunnyvale, California, this software package combined spreadsheet and word processing capabilities with several other functions and soon gave Ashton-Tate its second leading software program.

In 1983, a publishing group was added, making Ashton-Tate the first microcomputer software company to produce books that help computer users

understand and use hardware and software products. At about the same time, the company began to assert itself in the international marketplace by making its products available in foreign languages.

Tate died suddenly in August of 1984, and though the company mourned the loss of one of its founding fathers, it focused on his vision of growth and technological expertise as a mission for the future. In October of 1984, Cole left the company, and Edward M. Esber, Jr., then executive vice president of Ashton-Tate's sales and marketing operations, took over the helm.

Under Esber's direction, Ashton-Tate continued to grow, acquiring Forefront Corporation in July 1985 and introducing the second-generation Framework II in September of 1985. Building on the successful formula of Framework (which accounted for 20% of Ashton-Tate's revenues during the fiscal year ended January 31, 1985), Framework II enhanced the program's functionality and ease of use.

In November 1985, Ashton-Tate introduced dBASE III PLUS, the most complete data base management system for microcomputer users. The product improved upon the industry standards established by dBASE III by offering greater depth, power, and ease of use. That same month, Ashton-Tate relocated from its original Culver City, California headquarters to a new and larger facility in Torrance, California.

In December 1985, Ashton-Tate broadened its product line significantly with the acquisition of MultiMate International Corporation of Hartford, Connecticut. MultiMate is the producer of several word processing packages, including the best-selling MultiMate Professional Word Processor. The addition of MultiMate gave Ashton-Tate leading products in three software market segments—word processing, data base management, and integrated software—and propelled the company into its current position as the second-largest applications software company in the world.

At present, Ashton-Tate provides comprehensive service and support to small and large businesses and sells its products through an extensive distribution network, including retail dealers, value-added resellers (VARs), and hardware and software vendors. Now a leader in the international software marketplace, the company derives more than 25% of its total revenues from international sales.

PRODUCTS

dBASE III PLUS is the latest release in Ashton-Tate's dBASE family of products, the best-selling data base management software (DBMS) products in microcomputer software history. It is the successor to dBASE III, of which 300,000 copies have been sold between 1984 and January 1986. Its predecessor, dBASE II, was the first full-function relational data base management program for 8-bit microcomputers. Ashton-Tate has sold more than 325,000 packages of dBASE II since its introduction in 1981.

Ashton-Tate's dBASE III PLUS, for 16-bit microcomputers, addresses the needs of beginning and experienced computer users. It allows them to store, edit, retrieve, and print information quickly and easily across a broad range of applications—without programming. Powerful in a single-user configuration, its built-in multiuser capability satisfies the growth needs of businesses and organizations planning to link computers on a local area network (LAN).

dBASE III PLUS improves upon dBASE III by offering greater depth, power, and ease of use. dBASE III PLUS sorts up to two times faster and indexes up to 10 times faster than dBASE III. In addition, the Assistant, the dBASE III PLUS menu-driven user interface, employs pull-down menus with improved speed, performance, and power.

dBASE III PLUS offers beginning users a direct growth path to more productivity and power by providing easy access to the program's increased capabilities.

The improved data base product incorporates several features that increase the capabilities of the product without requiring programming by users. These include the Advanced Query System, which uses pull-down menus—not commands—for retrieval of specific information; View, which provides advanced relational capability through virtual connection between data bases; Screen and Applications Generator, which create custom applications without programming.

dBASE III PLUS is also a powerful tool for programmers developing specific applications. It incorporates more than 50 additions to the dBASE programming language, including RunTime+. It also includes the Data Catalog, which organizes and runs relational files.

dBASE III PLUS also has a built-in multiuser capability that allows simultaneous access to data by multiple users without fear of data corruption on a LAN. Users can also purchase the dBASE III PLUS LAN Pack, which contains three access disks that allow three additional users on a network to share dBASE III on a network-only basis.

dBASE III PLUS, in a stand-alone mode, requires only 256Kb of random access memory (RAM). To run on a LAN, the product requires a minimum of 640Kb of RAM at the server and 384 Kb at the workstation.

The suggested retail price of dBASE III PLUS is \$695, including the single-user DBMS, the dBASE Administrator, and one dBASE Access program. The dBASE III LAN Pack, sold separately, is priced at \$995 for three full sets of documentation.

dBASE II, the first full-function relational data base management system for microcomputers, quickly established itself as the standard for 8-bit computers. The program organizes and structures large volumes of data providing rapid access to specific information. It may be used in any business environment to handle a variety of information management needs, including maintaining data on, and generating reports relating to, accounts receivable, accounts payable, inventory, personnel, customers, payroll, and mailing lists.

dBASE II incorporates dBASE, the built-in applications development language that permits users to write programs for their specific information management needs.

With a suggested retail price of \$495, dBASE II runs on most popular 8- and 16-bit computers, including those from IBM and Apple.

Framework II, the integrated software package that combines the power of a spreadsheet with a word processor, was built on Ashton-Tate's successful Framework product introduced in July 1984. Framework was developed by Robert Carr, director of Ashton-Tate's Forefront Development Center.

Framework received widespread acclaim from the editorial and consulting community, including being named "software product of the year" in France, where it outsold even Lotus 1-2-3. Additionally, Framework won the PC Magazine spreadsheet Face-Off, was picked as the number one integrated

package in exhaustive benchmark tests conducted by *Software Digest Ratings Newsletter*, and was named one of the "most significant products" of 1984 by *PC Week* magazine.

Framework II broadens the standard by providing additional functionality and greater ease of use than its predecessor.

The seamless integration of spreadsheet and word processor in Framework II enables users to organize concepts and develop finished documents, complete with words, numbers, and graphs.

Framework II's spreadsheet is comparable to the best stand-alone spreadsheets and goes beyond their limitations by including a data base, graphics, telecommunications links, and programming, as well as tight integration with the word processor. Additionally, Framework II's word processor is equal to the most popular stand-alone word processors and includes outlining, spelling checker, mail-merge, and abbreviations, in addition to close integration with the spreadsheet.

Framework II's innovative user interface of pull-down menus provides consistent command structure throughout all functions and outlines for organizing thoughts and ideas. It also presents users with an intuitive means of breaking down information into manageable chunks that can be tied together in the form of spreadsheets, graphs, and text.

Framework II was developed to coexist with other productivity packages entrenched in the market. The product provides a variety of import/export formats, such as dBASE II/III, IBM DCA/Displaywrite, Lotus 1-2-3 (version 1.0), WordStar, MultiMate, ASCII text, and VisiCalc DIF.

Framework II runs on the IBM PC, PC XT, PC AT, 3270 PC, and compatible systems with 384Kb of RAM and two 360Kb floppy drives or a single 360 Kb floppy and a hard disk. Framework II's suggested retail price is \$695 and includes the spelling checker and outliner.

PRODUCT MARKETING

Ashton-Tate attained its position as a leading supplier of microcomputer software through a diverse marketing strategy. Products are distributed worldwide through various channels, including dealers, distributors, hardware, original equipment manufacturers (OEMs), VARs, and its own sales force.

Typically, small businesses and individuals purchase computer products through dealers. Hence, an integral part of the company's distribution strategy is its Authorized Dealer Program, which is designed to ensure that Ashton-Tate products are sold only through qualified dealers. For end users, this program guarantees the highest-quality product backed by Ashton-Tate's standards of service and support.

Currently, more than 3,500 dealers—including most of the major computer specialty retail chains—sell Ashton-Tate products nationwide.

A number of independent distributors also market the company's products. These groups typically market to retailers, offering them a range of products, merchandising, inventory, and support services.

In foreign markets, Ashton-Tate sells its products through distributors, sales representatives, and foreign subsidiaries. Presently, the company works jointly with major hardware manufacturers and has agreements with master distributors in several countries.

Ashton-Tate subsidiaries are located in the United Kingdom, the Netherlands, West Germany, Spain, and Australia.

Ashton-Tate serves the Japanese market through Nippon Ashton-Tate, a company formed under a joint venture agreement with Software International, a subsidiary of Japan Systems Engineering (JSE), Tokyo, Japan.

In addition to these channels, the company has a number of OEM and VAR agreements with independent software developers and the leading microcomputer manufacturers. Many of the software developers market dBASE applications to end users using RunTime, an Ashton-Tate package, which is a core version of dBASE II. RunTime provides developers with all the features necessary to create vertical market dBASE applications for industries such as insurance, oil, gas, and the dental/medical professions.

Ashton-Tate supports these channels with its own sales force in five regional offices: Northeast, with headquarters in New York City; Southeast, in Washington D.C.; Midwest, in Chicago; Southwest, in Dallas; and West, in Los Angeles. Branch sales representatives are also located in Boston, San Francisco, Denver, and Minneapolis.

The regional sales representatives are responsible for working with key dealers who sell to and service national accounts, some of which include the largest corporations in the world and major government agencies.

MULTIMATE INTERNATIONAL CORPORATION

Ashton-Tate acquired MultiMate International Corporation—producer of several word processing packages, including the best-selling MultiMate Professional Word Processor—in December 1985. The acquisition, which was the largest ever in the microcomputer software industry, made Ashton-Tate the second-largest independent microcomputer applications software company in the world.

MultiMate International Corporation, established in 1982, is the leading developer and marketer of microcomputer word processing software for the business arena. The company has dominated the corporate market by providing products that bring to the microcomputer the power and flexibility established by the previous standard—the Wang dedicated word processor. In addition, MultiMate has answered the unique needs of professional users by providing a comprehensive service and support organization and one of the first site-licensing programs in the microcomputer software industry.

The first product marketed by MultiMate was the MultiMate Professional Word Processor. The program has been adapted for a wide range of personal computers, has been translated into five languages, and has been enhanced considerably with a LAN version and advanced functions.

MultiMate Professional Word Processor 3.3 Series is a menu-driven package containing more than 130 text-editing and document-handling features, including an 80,000-word spelling checker, macros for creating boilerplate information, and a merge utility to exchange data with other software products.

MultiMate also produces the MultiMate Advantage Professional Word Processor, which improves upon the original by providing automatic footnoting, line and box drawing for creating charts and tables, and several additional advanced features. MultiMate Executive Word Processor is an easy-to-use version of Advantage with all of the same text-editing and document-handling features.

The company provides filing and graphics packages, On-File and Graph-Link, from which data can be merged directly with MultiMate documents.

The acquisition of MultiMate by Ashton-Tate is a reflection of Ashton-Tate's policy to acquire companies that offer top-quality products that meet a definite market need. MultiMate operates out of its East Hartford, Connecticut headquarters as a wholly owned operation of Ashton-Tate. MultiMate reported sales of \$21 million in its last fiscal year ended March 31, 1985.

SERVICE AND SUPPORT

Ashton-Tate's responsive service and support programs are key elements in the company's success. Through high-quality documentation, telephone support, and user training programs, the company provides a level of service and support unrivaled in the industry.

Ashton-Tate's commitment to service is reflected in its unprecedented creation of a senior management position to head its documentation and support organization. Robert D. Gafford, vice president of consumer communications, manages a staff of more than 100 in four major areas: product documentation, testing, telephone support, and customer relations.

Through its Corporate Emphasis Support Program, Ashton-Tate provides an extensive level of product and technical support and training for corporate users and dealers who sell to that audience. Primary elements in the program include on-site assistance from sales engineers, a dedicated telephone support line, a nationwide training program, and support for corporate users.

With hundreds of thousands of people using its dBASE, Framework, and MultiMate product lines, Ashton-Tate receives an average of 900 technically oriented telephone inquiries each day. The company has 35 expert technical support people to answer these inquiries. On average, with this dedicated hot line, users receive answers to their questions within 2 1/2 minutes.

The company electronically distributes support-related information, using three on-line "bulletin boards": CompuServe and The Source, to general users, and One Point, to dealers and corporate customers. Information and technical tips are updated weekly.

Ashton-Tate has also developed a series of on-disk tutorials—electronic documentation that provides an interactive method by which users can increase their mastery of their software. On-disk tutorials are included with dBASE III PLUS and Framework II. The dBASE III PLUS documentation includes a volume that teaches users how to program, in addition to a "Getting Started" booklet and a "Learning and Using" dBASE III PLUS volume.

PUBLISHING GROUP

In 1983, Ashton-Tate was the first microcomputer software company to establish a publishing group. Since its inception, the group has published more than 40 high-quality products on such diverse topics as dBASE III PLUS, Framework II, telecommunications, documentation, UNIX, Pascal, and the computer industry.

The group's publications include *Everyman's Database Primer for dBASE III PLUS*, *dBASE II For Every Business*, *Framework II: An Introduction*, and the *Through The MicroMaze* series.

The group also publishes a quarterly magazine called *The Ashton-Tate Quarterly*, which provides Ashton-Tate customers with practical advice on how to use the company's products.

The publishing group also offers book/disk packages and add-in software to help computer users to understand and use their software better.

INTERNATIONAL OPERATIONS

With many foreign language versions of its products, a solid distribution and sales network, and the early success of dBASE II, Ashton-Tate quickly established itself as a leader in the global market. International sales accounted for 28% of the company's \$210.8 million in revenues for the fiscal year ended January 31, 1987, and the company expects continued growth in foreign markets.

Among the foreign language versions of its products, dBASE II has been translated into 11 languages, including Dutch, Swedish, and Japanese. Introduced in Europe during the summer of 1984, dBASE III and Framework are available in 10 languages, including Italian, French, German, and Spanish. Introduced in Europe during the fourth quarter of 1985, Framework II and dBASE III PLUS will also be translated into a variety of foreign languages.

Ashton-Tate sells its products internationally through a comprehensive network of distributors, dealers, and microcomputer manufacturers. Subsidiaries in the United Kingdom, the Netherlands, West Germany, Australia, and Spain service the company's international customers, whereas master distributors in several countries, including France and Italy, and major hardware manufacturers, including IBM, Olivetti, ACT, and Ericsson, round out the multichannel distribution system.

Nippon Ashton-Tate develops and sells software products specifically designed for the expanding Japanese market.

International operations for Australia, New Zealand, Latin America, and developing territories are managed from the company's Torrance, California, headquarters, and the company's international headquarters for European operations are located in Maidenhead, England. Far East operations are headquartered in Tokyo, Japan.

MANAGEMENT

A key element in Ashton-Tate's industry leadership was the formation of an experienced management team. Since assuming his position as chairman and chief executive officer in August 1986, Edward Esber, Jr., has guided the company on its growth path. Esber joined Ashton-Tate in May 1984 as executive vice president, marketing and sales. Previously, he held several executive positions at VisiCorp, was a marketing manager for Texas Instruments' Consumer Products Division, and held engineering positions with IBM's systems Products Division.

Esber has an M.B.A. from Harvard Business School, a M.S.E.E. from Syracuse University (New York), and a B.S. in computer engineering from Case Institute of Technology in Ohio.

The company's three other executives are Luther J. Wussbaum, president and chief operating officer, Roy E. Folk, executive vice president of the Software Products Division, and George Farinsky, executive vice president, finance and chief financial officer.

ASHTON-TATE, INC.

ASSEMBLY LANGUAGE AND ASSEMBLERS

INTRODUCTION

Assembly language is a symbolic language used to communicate with a computer at the level of the hardware. It is the most basic language used to create programs other than machine language itself. The machine responds to patterns of binary digits, which are actually voltage or current levels in the operating computer. For example, the pattern 0010 0001 might be interpreted as ADD (0010, on the left) the contents of memory location 1 (0001, on the right) to the value of the accumulator (not named, but implicit in the instruction on the machine described below), and place the result of the addition in the accumulator (also implicit).

One could enter the instruction directly, for example, through switches on the front panel of the machine, if one had access to the actual bits of the memory or processing unit of the computer. However, the process is tedious for programs consisting of several instructions, so a machine code that abbreviates the bit patterns was developed. One example is the hexadecimal (hex) code, in which the patterns are grouped into sections of 4 bits each, with position values 8, 4, 2, and 1. The hexadecimal code for the above pattern would be 21. Given the hexadecimal machine code, one could type 21 on the input device, for example, the keyboard, and the program interpreting the machine code, the "nonsymbolic assembler," or machine code interpreter, would place the bit pattern 0010 0001 in a specified memory location.

Using machine codes like 21 for "add the contents of memory location 1 to the accumulator" is somewhat simpler than working with the bit patterns; but if there is a large number of instructions, the instruction codes (left-hand side) and the memory locations used (right-hand side) can be difficult to remember. For this reason, symbolic assembly language was developed. Using a symbolic assembly language, one writes statements such as ADD B, and the assembler translates these statements into the bit patterns 0010 0001. For ease of reference these patterns are sometimes summarized as hexadecimal (or octal or decimal) digits, for example, 21, when printing the contents of main memory or other locations (e.g., the accumulator) of the computer.

An assembler is simply a program, usually provided by the manufacturer of the computer, or a software house, that translates the assembly language—a phrase that will henceforth be taken to be synonymous with "symbolic assembly language," or "assembler"—into machine code. It is this close relationship of the assembly language to the machine code that makes the study of assembly language and assemblers an excellent introduction to the architecture and operations of a computer, the hardware (the machine itself), the systems software (the assembler or translator program), and the applications software (the programs you write).

1.1 THE BASIC MACHINE

To explain the concept of assembly language and an assembler, one needs to have a basic machine in mind. The machine visualized here is that of Figure 1.1.

a. PROCESSING UNIT

Registers

AC

PC

IR

IC

MAC

Flags

P

N

Z

Devices

Adder

Comparator

b. MAIN MEMORY

1

2

3

4

c. PERIPHERAL DEVICES

Input Device

Keyboard

Output Device

CRT Screen or Monitor

FIGURE 1.1 The basic machine. (a) Processing unit; (b) main memory; (c) peripheral devices.

There are three basic sections to the computer system: the processing unit (Fig. 1.1a), the main memory (Fig. 1.1b), and the peripheral devices (Fig. 1.1c). The processing unit contains certain storage locations, called registers, for example, the accumulator (AC) and the program counter (PC), and certain "devices," the units used to perform calculations (adder) and comparisons (comparator). The word devices is enclosed in quotes because the devices included in the processor are combinations of circuitry on a microprocessor chip, not devices in the sense of a line printer, that is, not external devices.

The storage locations in the processing unit, or registers, have specific functions. The AC holds data values that are to be operated on by the adder or comparator, and the results of these operations. The PC holds the address of the next instruction to be executed. The instructions in a program are stored in the main memory of the computer and "fetched" when it is time for them to be executed. The contents of the PC indicate, to the control unit of the processor, what instruction is to be fetched next. The PC and the AC will be the primary registers of interest, because the one (AC) is involved in all computations and comparisons, and the other (PC) is involved in the execution of every instruction.

The instruction register (IR) is the register used to hold an instruction when it is fetched from the main memory. It is used to decode the instruction into the instruction code (IC), which is the first part of the instruction, and the memory address code (operand address [MAC]), the address of a storage location in main memory, the second part. These portions of every instruction, the IC and the MAC, correspond to the left-hand part, ADD or 2, and the right-hand part, B or 1, discussed above. The IC tells us what to do (ADD). The MAC tells us "with what"; it gives the address of the location in main memory that holds the data value to be used in the calculation or other operation, location 1.

The flag registers are used in response to the outcome of certain calculations. The flags shown indicate whether the result of an addition, subtraction, or comparison was positive (P), negative (N), or zero (Z). The flags are "set" (to one) if the answer to the question they pose is yes and are cleared (set to zero) if the answer is no. Hence, if a calculation resulted in a positive nonzero result, the flag values would be $P = 1$, $N = 0$, and $Z = 0$.

The main memory is used to hold the programs and data utilized in operating the computer. These include the systems programs, for example, the operating system (PC-DOS or MS-DOS for many microcomputers) and the assembler; BASIC, Pascal, or other interpreters and compilers; packages for word processing, spreadsheet, data base management, communications, and graphics applications; and the applications programs, the programs we write, in the languages or with the packages.

We will be interested in applications programs written in assembly language and the systems program, called an assembler, which translates those assembly language programs into machine language, as well as the means of executing the machine language program produced by the assembler.

Finally, we assume two peripheral devices, an input device (the keyboard) and an output device (the monitor). These will be used to enter data into the computer and obtain results from the computer. The examples shown here do not require secondary storage devices, but if they did, this portion of the diagram would be their logical location.

NOTE: The machine described here is a single AC machine. Other machines will be discussed, more briefly, later in the paper. The rationale for using a single AC machine is that it simplifies the construction of the assembler, yet the concepts are applicable to machines with several ACs or registers.

1.2 A SAMPLE PROGRAM

The easiest way to begin is to use a sample assembly language program. The program is written in a pseudoassembler code developed for the conceptual machine of Figure 1.1, which is called ENC. The program is shown in Figure 1.2.

An assembly language statement consists of four fields: label, opcode, operand, and comment. The label field and comment are optional; the opcode and operand are not, although some statements, for example, HLT in Figure 1.2, may not require an operand.

The Opcodes

The opcode tells the computer (central processing unit) what to do; it consists of commands such as LDA (load accumulator), ADD (add to the accumulator), STA (store the contents of the accumulator), and HLT (halt). These commands are provided by the manufacturer and are listed in the reference manual for the machine. A subset of the opcodes (operation codes) for our machine is given in Figure 1.3.

Figure 1.3 gives both the machine code and the symbolic name for that machine code. We will write the programs in the symbolic code, and the assembler will translate the symbolic code into the machine code. The symbolic code is mnemonic in that the combinations have been chosen to reflect the meaning of the codes (LDA for load the accumulator; STA for store the accumulator; ADD for add to the accumulator; HLT for halt.). The remaining commands listed in Figure 1.3 are INP (input), OUT (output), and SUB (subtract).

The Operands

The operands give the addresses of the locations to be used in the commands. For example, in the command LDA A, the AC is to be loaded with the contents of the memory location, called A. In the command ADD B, the data value in the location called B is to be added to the contents of the AC and the result placed in the AC. Finally, in the instruction STA C, the contents of the AC are to be stored in the memory location C.

```

A: 5
B: 6
C: RES 1
START: LDA A      ; (MEM) → (AC)
        ADD B      ; (AC) + (MEM) → (AC)
        STA C      ; (AC) → (MEM)
        HLT
        END START

```

FIGURE 1.2 Sample assembly language program.

Reference Manual ENC Machine

Machine Code	Symbolic Opcode
0	INP
1	OUT
2	ADD
3	SUB
4	LDA
5	STA
6	HLT

FIGURE 1.3 Subset of opcodes.

The Data Locations

The memory locations referred to as A, B, and C in the instructions are those labeled A, B, and C in the program.

```
A: 5
B: 6
C: RES 1
```

The contents of A are defined to be a 5, A: 5, the contents of B are defined to be a 6, B: 6, and the contents of C are undefined, C: RES 1.

One creates a storage location for the data by labeling it A: B: C:, and either storing an actual data value,

```
A: 5
B: 6
```

or just "reserving" space, C: RES 1. The statement RES stands for "reserve space" (a word or byte of memory; in our case, a word); and the number 1 tells the assembler how much space to reserve. RES is not a command to the computer (the processing unit, which is hardware) but a command to the assembler (the program, software, unseen but present—it is the program that is translating our program, Fig. 1.2). One distinguishes commands to the processing unit ("real" opcodes, like ADD or LDA) from directives to the assembler (like RES) by calling the latter "pseudo-ops." They are like opcodes in that they appear in the opcode field (RES is in the same field as LDA, the second or opcode field in the four-field format), but they are pseudo in that their effect takes place when the application program is translated (at assembly time) not when it is run (not at execution time).

Define Constant Versus Define Storage

The use of a directive, such as A: 5, does two things: (a) it reserves a memory location for the data value 5, and (b) it places the value into this location. This is referred to as defining a constant (DC). Directives such as C: RES 1 define storage (DS). They reserve space, but they do not place a data value into that space. The value of the space is undefined in the sense

that it could be any value (either zero if the assembler zeros it out or "garbage" if it does not). Such locations are used to receive the results of calculations, as C will in the program above, or data values input from an external device (shown below for the keyboard).

Labels

In the following assembler statements,

```
A: 5
B: 6
C: RES 1
```

the fields A:, B:, and C: are labels. (The term "statement" is the general term applying to both real instructions, like LDA A, and directives, like RES.) The label has a value, A, which serves the same purpose as a variable name in a higher level language when the label is used to identify a data location (as A, B, and C do in Fig. 1.2). One refers to the location or "variable name" in the statements referring to (involving) that location: ADD B. The label values (names) can vary in length up to some maximum number; the colon indicates the end of the label field.

Labels can also be used to identify memory locations holding instructions. An example is START: LDA A. The contents of this memory location, when the program is stored in the main memory of the computer, will be LDA A. Or, more precisely, the content will be the machine code for LDA A, which, in the computer ENC, will be 40 in hexadecimal notation, or 0100 0000 in binary notation.

Labels used to identify memory locations holding instructions are analogous to the labels used in the same manner in higher level languages, for example, the line number 10 in the FORTRAN and BASIC instructions,

```
10 Read (5,15) A
10 Read A
```

or the paragraph name in the COBOL statement,

```
PARA-A.
  Read...
```

These labels are the object of GOTO statements (in BASIC and FORTRAN) or PERFORM statements (in COBOL):

```
GOTO 10
PERFORM PARA-A
```

They are the object of jump statements in assembler (ASM): JMP START. The jump is the assembly language equivalent of a GOTO. (The jump statement and its variations, for example, "jump less than," are described below.)

The END Directive

In the program of Figure 1.2, the label **START** is referenced in the directive (pseudo-op) **END START**. The **END** directive has two functions:

1. It demarcates the physical end of the program.
2. It indicates the starting address of the program (for execution).

Assembly language programs are variable in length. The assembler must have some mechanism to know where they end. The mechanism utilized is the same as that used in the movie business: The word **END** appears on the last physical line of the program.

Execution of an assembly language program can start with any line. The name of the location with which to start execution is given in the **END** statement. In this program we have used the label **START** in order to be mnemonic: Execution begins at **START**. However, it is not the content of the label but its appearance on the **END** statement that causes the assembler to know where we intend execution to begin. If we had written **END A** on the **END** statement of Figure 1.2, execution would begin on the line labeled **A**, with disastrous (or at least unexpected) results, because **A** holds what was intended to be a data value, 5, not an instruction.

Data Versus Instructions

There is no inherent distinction between data and instructions at the machine level. Both appear as bit patterns. The bit pattern for the data value 5 is 0000 0101 in our machine. The bit pattern for **LDA A** is 0100 0000.

One cannot tell whether a given bit pattern is meant to be a data value or an instruction by looking at the bit pattern. For example, in the pseudomachine, **ENC**, the bit pattern for the data value 5, 0000 0101, can also be interpreted as the instruction, "Input data into location 5."

One distinguishes the data values from the instruction values by "keeping them out of the way" in the program one writes. In Figure 1.2 this was done by placing the data values (**A**, **B**, and **C**) above the executable portion of the program (from the line labeled **START** to **HLT**).

In terms of the machine, it tells the difference between instructions and data values by the locations in which the bit patterns occur. Any bit pattern pointed to by the **PC** is considered to be an instruction. It will be fetched into the **IR** and decoded by putting the left part of the bit pattern into the **IC** (opcode) register and the right-hand part (memory address or operand part) into the **MAC** (in our machine usually memory address register [**MAR**] in other machines). Anything that terminates in the **MAC** is considered to be an address of either a data location or an instruction. (It is strictly the address of data; however, the distinction between locations holding intended data and the locations holding intended instructions is useful even in discussing the contents of the **MAC**.) Whether it is the address of a data location (**A**, **B**, or **C**) or a location holding an instruction (**START**), is determined by the opcode. Opcodes involving data values, such as **LDA**, **STA**, and **ADD**, force the data location interpretation. Opcodes such as "jump to location **x**," **JMP START**, force the memory location or "instruction" interpretation.

NOTE: The contents of the **MAC** give the address of the operand for the instruction. Sometimes this is a data location, the contents of which

are to be fetched—a data transfer. Sometimes it is simply a location, an address that will be placed in the PC to cause a transfer of control without a data fetch. This is explained more fully below.

The Comments

The comments are for documentation purposes; they have no effect on the execution of the program. They do, however, illuminate the purpose of the code, which is usually more obscure at the assembly language level than in programs written in higher level languages.

The comments shown in Figure 1.2 are intended to convey what the instructions do in the symbolism found in reference manuals. The notation (MEM) \rightarrow (AC) indicates that the contents of the memory location (A in LDA A) are to be loaded into (become the value of) the AC (LDA in LDA A). This is sometimes written as (MEM) \rightarrow AC, with the left-hand side indicating "contents of" (parentheses) and the right-hand side indicating an address (no parentheses). However, we have chosen to parenthesize both sides, (MEM) \rightarrow (AC), to indicate the statement "the contents of the memory location becomes the contents of the accumulator."

A "load" moves data from memory to the AC. A "store" goes in the opposite direction: (AC) \rightarrow (MEM). The contents of the AC become the contents of the memory location, for example, B in STA B. An arithmetic instruction has the form (AC) op (MEM) \rightarrow (AC). The contents of the AC (left-hand side) are to be "operated on" (op: add, subtract, multiply, divide, etc.) with the contents of a memory location (MEM), the result being placed back into the AC (right-hand side). For the opcode ADD, we have (AC) + (MEM) \rightarrow (AC). For SUB, we have (AC) - (MEM) \rightarrow (AC). Note that the memory location is always the second operand in these binary operations, at least in our machine, ENC.

Each machine has its own set of opcodes and its own conventions for interpreting the instructions, for example, LDA A, or directives, for example, A: 5, C: RES 1, or END START. Hence, when dealing with assembly language, one must utilize the reference manuals for the machine and texts written for that particular machine. However, all assemblers have certain elements in common, as do all machines. For example, almost all assemblers follow the four-field format: label, opcode, operand, comment. And almost all machines have special registers, such as the PC and the IR; an AC or a set of ACs (called the registers: arithmetic, index, and other types); internal devices, such as an adder and a comparator; and external devices, such as the keyboard, monitor, and secondary storage devices (disk and tape drives). We will first address some of the issues common to virtually all assemblers and machines and then discuss some of the distinctions among various assemblers and machines at the end of the article. At this point, we want to address the execution and translation of the assembly language program. The translation precedes the execution in temporal order, but we treat the execution first because it helps enlighten the translation process.

1.3 EXECUTION OF THE SAMPLE PROGRAM

When the program of Figure 1.2 executes, the first instruction executed will be LDA A. This will cause the contents of the memory location A, a 5, to be placed in the accumulator, A: 5 \rightarrow AC: 5. The next instruction executed will be to add the value of B (6) to the value in the AC, placing the result in the AC, AC: 5 + B: 6 = AC: 11. The addition is performed by sending the two input values, 5 and 6, to the adder and placing the result of the addition in the AC:

```
B: 6  $\rightarrow$ 
AC: 5  $\rightarrow$  adder  $\rightarrow$  AC: 11
```

The adder is being treated as a "black box" in our discussion, that is, we are not concerned with its internal architecture. One can, however, find treatments of the subject in books on computer hardware design [1,2].

The next instruction, STA C, takes the value 11 in the AC and stores it in the memory location C, AC: 11 \rightarrow C: 11. We cannot see the value of C, but it can usually be printed from the internal memory through a routine known as a "dump," which dumps the values of the registers and any main memory locations specified, or a "debugger," which allows one to trace the action of a program, as well as to enter data values and instructions into memory and print the values in various memory locations. Discussions of these routines can be found in the reference manuals and, usually more clearly, in texts. For example, a good treatment of the assembly language for the IBM PC, a treatment that utilizes the debugger to show the "internals" of the machine, can be found in Ref. 3. After storing the result of the addition in C, the program then executes the instruction HLT, which halts execution of the program.

HLT and END

HLT is an instruction that causes execution of the program to stop. END is a pseudo-op, a directive to the assembler, which causes translation of the program to stop. The HLT instruction can be placed anywhere, physically, in a program. (It should be placed, logically, where one wants the execution of the program to terminate.)

The HLT instruction is usually implemented as a call to a routine in the operating system. For example, on the IBM PC it is implemented as the instruction INT 20, which is translated into machine code as CD 20. The INT instruction "interrupts" the operation (execution) of our program (the program of Figure 1.2) and transfers control to the operating system (CD: change to DOS). The operating system then has to decide what our program has asked it to do. The code, 20 for HLT, tells it what we want. For the IBM PC, the code 20 means "end of execution of the program." Other codes, for example, 21, mean output a data value or input a data value. These codes are also explained in the reference manuals and texts [3,4].

Interrupts

An interrupt is simply a mechanism for communicating among programs: application programs and the operating system, in our case. For example, the operating system routines usually do the actual inputting and outputting of

data values from and to devices. They do so in response to a request from the application program. This is illustrated in the next section.

Input and Output of Data Values

A program that inputs data values from an external device and outputs data values to an external device is shown in Figure 1.4. Execution of the program begins on the line labeled EX (for execution). The program first inputs a data value from the input device, in our case the keyboard, and places the value in the location A. The format of the instruction is "input (from device) (to location) A." The location has been reserved for the data value, A: RES 1. The "define storage" form of reserving memory has been used because the input command will cause the data value to be stored in the location specified (in the INP command).

As indicated above, input and output commands are calls to the operating system. The call to input or output, in IBM PC assembler, is INT 21. The INT is the interrupt command, and the value 21 is a code that tells the operating system that a "read" or "write" is requested. A further code is required on the PC to determine whether the request is to read (R) or write (W), to determine which device the read or write addresses, and to determine how much to read or write (e.g., a character of a string). The code to read a character on the PC from the terminal keyboard is 1; the code to write a character to the monitor is 2. Other codes are used to choose other devices, or to read and write strings of characters. The values must be placed in a special register, AH, prior to the issuance of the INT 21 command (AH stands for the high end of the A register).

The exact protocol (convention) for communicating with the operating system varies from machine to machine, but the general procedure is the same: (a) Call the operating system (INT), and (b) tell it what you want it to do (code 21 = R/W; and AH = 1 → R: keyboard, character). The operating system then fulfills the request (if it is a legal request). In our case, INP A, the request is legal. The operating system performs the read, transfers the data value to the user data area (the location A), and returns control to the user program.

The second command executed is also an input command, this time inputting the data to location B. The next portion of the program is the same as the first program: The value in A is loaded, the value in B is added to it,

```

A:   RES 1
B:   RES 1
C:   RES 1
EX:  INP A
      INP B
      LDA A
      ADD B
      STA C
      OUT C
      HLT
      END EX

```

FIGURE 1.4 Program inputting and outputting data values.

and the result is placed in location C. However, in this program the result is output to the screen—INT 21 in the IBM PC assembler (with AH = 9, because the result, 11, when output, will be a string, "11").

The advantage of this program over the first program is that the data values are entered more flexibly. In the first program the DC statements A: 5, and B: 6 act like assignment statements or data initialization statements in higher level languages. (They are, at the assembler level, data initialization statements; the loads and stores, called "moves" on some machines, are assignment statements.) To execute the program with other data values, we would have to change these lines. In the program of Figure 1.4, the input statements accept any data values the user of the program inputs; the result is then computed and output.

In the program of Figure 1.4, it is assumed that the data values are placed into the memory locations, A and B, in binary form. This would not be the case in an actual assembler program. The data values input from the keyboard are coded in ASCII format. For example, the number 5 would be coded as 065 (in octal notation) or 0110101 (in binary notation) in ASCII.

A routine called "convert ASCII to binary" (cvtab), either supplied by the manufacturer or software house, or written by the local computer center or the application programmer, would have to be executed before arithmetic could be performed on the number. The routine essentially "strips off the sixes" in numbers entered as 065062 but operated on as 52 (065 → 5; 062 → 2). Similarly, on output, the binary result in C would have to be converted to ASCII ("cvt bin to asc," or cvtba) by an inverse routine. We have abstracted from these issues in the program of Figure 1.4 because it would clutter the program unnecessarily. The routines are well known and are explained and illustrated with code in standard assembly language texts [5, 6].

The JUMP Statement

The program of Figure 1.4 executes once. For it to execute more than once, we would include JUMP statements, as shown in Figure 1.5. The JUMP statement JMP BEGIN is an unconditional jump. It transfers control from the location that would have been executed next (the HLT) if no jump had occurred—to some other location, here the location labeled BEGIN.

The unconditional jump (JMP) acts like a GOTO in higher level languages: GOTO BEGIN. It is called an unconditional transfer of control, because the transfer is always made.

Conditional Transfer of Control

A conditional transfer of control is indicated in the program of Figure 1.5 by the sequence of statements

```
CMP
JZ QUIT
```

The compare instruction (CMP) in our machine examines the value in the AC and sets the flags accordingly: If the value is zero, it sets Z to 1; if positive and nonzero, it sets P to 1; and, if negative, N to 1. The next instruction JZ QUIT tests (examines) the values of these flags. If the test is "true," that is, if the zero flag is set when testing JUMP ZERO (JZ), the jump to QUIT is executed. If the zero flag is not set, that is, the condition is not satisfied or "false," the jump is not taken; the next statement (in physical sequence) is executed, in this case, INP B.

```

A: RES 1
B: RES 1
C: RES 1
BEGIN: INP A
        LDA A
        CMP
        JZ QUIT
        INP B
        ADD B
        STA C
        OUT C
        JMP BEGIN
QUIT:   HLT
        END BEGIN

```

FIGURE 1.5 JUMP statement.

The sequence of statements

```

CMP
JZ QUIT

```

acts like an IF statement in a higher level language. For example, in the program of Figure 1.5, the sequence acts like the statement IF (A = 0) THEN GOTO QUIT. This is a result of the entire set of code

```

INP A
LDA A
CMP
JZ QUIT

```

The value of A is input and is then loaded into the AC. The value in the AC (A's value) is then tested (CMP) for positiveness, negativeness, or zero value, and the flags are set. Finally, the flags are examined (Z in JZ) to see whether or not a jump (J in JZ) should be taken. In this case, the condition is "jump if zero." Other jumps would be jump positive (JP) and jump negative (JN) in this scheme.

The CMP shown here is one of the simplest that can be implemented. Other CMPs would compare the value in the AC against a data value stored in memory, CMP B, or, in machines with more extensive operands, one could compare the values of two memory locations, CMP A,B. However, the intent of all the schemes is to implement a conditional jump. The mechanism is to perform some test that sets the flags, CMP, and then jump (or not) according to the value of the flags set.

In the program of Figure 1.5, the intent of the test is to allow the user of the program to have a mechanism for stopping execution. If the user wishes to quit, he or she enters a zero for the value of A. This convention would have to be established ahead of time, either in the documentation accompanying the program or in instructions printed out when the program is run.

Summary of Execution

In this section we have examined the form and execution of some sample assembly language programs. The execution is similar to that of other high-level language programs, utilizing the same type of commands (READ, WRITE, ADD, GOTO, IF). The form of the commands is, however, a little more detailed. For example, the single higher level language assignment statement $C = A + B$ requires the three assembler statements

```
LOAD A
ADD B
STORE C
```

A more complex statement, such as $D = (A + B * C) / 2$, would require the sequence of statements

```
LOAD B
MUL C
ADD A
DIV 2
```

It is the job of a compiler or interpreter to take statements written in the higher level language, such as FORTRAN, Pascal, or BASIC, and translate them into the assembly or machine language of a particular machine. A good portion of the compiler is involved in analyzing arithmetic expressions (in assignment statements) and the conditions in IF statements. The read and write commands translate directly into calls to the operating system, and the GOTOs (if they exist) are jumps.

The simplest form of a translator is an assembler. It translates the higher level language assembler into the lower level language machine. We turn to that process now. The assembler discussed here is a "two-pass" assembler.

NOTE: Assembler is a high-level language relative to the low-level machine code. Other programming languages, such as C or Pascal, are high level relative to assembler.

2.0 TRANSLATING THE PROGRAM TO ASSEMBLER

There is a distinction between assembly language, in which one writes programs, and the assembler, which translates these programs into machine language. The preceding section showed sample assembly language programs. This section examines the assembler, that is, the translator program, itself. The discussion will assume that the program is that of Figure 1.2, reproduced here as Figure 2.1, for convenience of reference.

The job of the assembler is to translate the symbolic code of Figure 2.1 into the machine code of Figure 2.2. It does so in two passes, referred to as "pass one" and "pass two."

```

A: 5
B: 6
C: RES 1
START: LDA A      ; (MEM) → (AC)
      ADD B      ; (AC) + (MEM) → (AC)
      STA C      ; (AC) → (MEM)
      HLT
      END START

```

FIGURE 2.1 Sample assembly language program.

2.1 PASS ONE: PRODUCING THE SYMBOL TABLE

Pass one of the assembler creates a symbol table consisting of the labels used in the program and their physical location. It does so by setting a line counter (LC) to zero and examining each line of the program (a pass through or over the program) to see if that line has a label, bumping (adding 1 to) the LC counter as it goes along.

For the program of Figure 2.1, the LC is initially set to zero, LC: 0, and line 0—the line containing the directive A: 5—is examined for the presence of a label. There is a label, A, so an entry for this line is made into the symbol table ("label table"):

Label	Value	(Address or location of the label; the number of the line on which it occurs)
A	0	

The LC is then bumped to 1, LC: 1 (LC: = LC +1), and the second physical line (line number 1 in the zero-based counting) is examined for a label. There is a label, B, so this symbol and its value (its address or line number) is entered into the symbol table:

Label	Value
A	0
B	1

The value of the LC is bumped to LC: 2, and the next line is examined, resulting in the entry for C:

Label	Value
A	0
B	1
C	2

The LC is then bumped to LC: 3, and the next line is examined, resulting in the entry for START. The entire symbol table now looks like that of Figure 2.3. The assembler would continue to examine the lines of code, bumping LC as it went along. For example, on line number 4 (physical line 5), it would find the command ADD B. This line has no label, so no entry would be made for it in the symbol table; likewise for the lines numbered 5 and 6,

```

0: 0 5
1: 0 6
2: 0 0
3: 4 0
4: 2 1
5: 5 2
6: 6 0

```

START: 3

FIGURE 2.2 Machine code.

```

STA C
HLT

```

The procedure ends with the END statement, which signals the physical end of the code, the point at which translation of the code should stop (because there is no more code to translate; the physical end of the program has been reached).

The basic procedure is

1. Set LC to zero.
2. Repeat until end.
 - a. Check line for label
 - b. If label
 - i. Enter label in label table
 - ii. Enter value of LC in label table; this is the address (line number) at which the label was found and will be used to reference that memory location.
 - c. Bump LC

NOTE: If no label is found, no action is taken; however, in both cases, label or not, the LC is bumped. It assigns the addresses, in physical order, counting from zero on up.

A program that illustrates the process with some variety is that of Figure 2.4. The line numbers have been provided in the listing so we do not have to discuss the bumping of the LC; however, it is by the bumping of the LC that the numbers are generated in the assembler.

Symbol	Table
A	0
B	1
C	2
START	3

FIGURE 2.3 Symbol table.

```

0: BEGIN   LDA X
1:         ADD Y
2:         STA Z
3:         HLT
4: X: 5
5: Y: 6
6: Z: RES 1
7:         END BEGIN

```

FIGURE 2.4 Data after code.

The symbol table for the program of Figure 2.4 will be that of Figure 2.5.

There is no requirement that the data (X, Y, and Z) appear at the beginning of an assembly language program. The only requirement is that data be kept out of the line of instructions (lines 0 through 3 in Fig. 2.4). This is commonly done by placing the data before the first executable instruction, as in Figure 2.1, or after the last executable instruction (HLT in Fig. 2.4), but before the END statement (or, in more advanced schemes, having the data loaded in a separate segment of memory from the instructions). (What would happen if the data were placed after the END statement? It would not be translated and, hence, would be unavailable to the executing program, because END signals the physical end of the code to be translated.)

The reader should verify that the symbol tables for the code of Figures 1.4 and 1.5 would be those shown in Figure 2.6 (a,b).

The actual operation of pass one is slightly more complex in that it must deal with the DC and DS directives. If the assembler finds a numeric constant after the label (in the opcode field, a pseudo-op of sorts), it must store this value in that location, for example, for the statement A: 5, it must create the value 05 and enter it on that line. It can do this here or in pass two. It is actually simpler for us to defer the action to pass two.

The RES statement does not require a data value to be stored, although some assemblers zero out any locations allocated in this manner.

NOTE: The IBM 360 did not zero them out; the DEC/VAX systems did.

Actually, the VAX zeroed them out sometimes and at other times did not. It did for statements similar to RES, A: .BLKW 10. It did not for statements similar to A: 5, A: .LONG, although one could include the zero in the declaration A: .LONG 0, which would be equivalent to C: 0 in our assembler. The point is, however, that the concepts common to machines of the von Neumann architecture are similar

Symbol Table	
Symbol	Address
BEGIN	0
X	4
Y	5
Z	6

FIGURE 2.5 Symbol table resulting from program of Fig. 2.4.

a	
Symbol	Address
A	0
B	1
C	2
EX	3

b	
Symbol	Address
A	0
B	1
C	2
BEGIN	3
QUIT	12 (or C, the hexadecimal symbol for decimal 12)

FIGURE 2.6 Symbol tables for program of Fig. 1.4 (a) and Fig. 1.5 (b).

across a long span of machines, from the earlier models, such as UNIVAC I, UNIVAC 1108, IBM 1401, IBM 1130, and IBM 360, to the more modern machines, DEC-10 (relatively modern, as it, too, is now historical), the PDP series, including the VAX, and the microcomputers, such as IBM PC. The history and architecture of von Neuman machines can be found in the historical literature [7-10] and the technical literature [11].

If one has coded the program correctly, it is immaterial whether or not the locations are zeroed out, except from an aesthetic point of view, because the intent is to have these data values "set" by a store or input instruction.

Arrays

The RES brings up one more issue—the allocation of more than one location. If one wishes, one can allocate a set of locations, D: RES 10, which is the equivalent of setting aside space for an array at the assembly language level, the analog of the higher level dimension statement Dimension D(10). If one does direct the assembler to set aside an array of locations, the LC must be bumped by that number, 10, in this case, D RES 10, instead of 1. This is shown in Figure 2.7(a-c). The code (Fig. 2.7a), causes the address for A to be zero, B to be 1, but C to be 11 (Fig. 2.7b). This has the effect of having generated 10 locations for B (Fig. 2.7c).

2.2 PASS TWO: THE TRANSLATION OF THE CODE

Pass two of the assembler uses two tables, the opcode table and the operand table (symbol table), to produce the machine code for the program.

a

```

A: RES 1
B: RES 10
C: RES 1

```

b

Symbol	Table
Symbol	Address
A	0
B	1
C	11

c

```

0: (A)
1: (B, the array)
2:
3:
4:
5:
6:
7:
8:
9:
10:
11: (C)

```

FIGURE 2.7 Program to set aside an array of locations.

The Opcode Table

The opcode table is a list of the opcodes (Symbolic) and their machine code equivalents (Fig. 2.8).

The opcodes are supplied by the manufacturer of the computer (ENC Incorporation, in our case) and are the codes one uses to write assembly language programs for that computer. The opcodes are referred to as the instruction set of the machine. (As the reader may have inferred, assembly language code is machine specific, not portable to a computer with a different architecture and instruction set.)

The Operand Table

The operand table is the symbol table that was produced by pass one. For the program of Figure 2.1, it is the symbol table of Figure 2.3. The symbol table is program specific, that is, it varies from program to program (the opcode table does not).

The production of the symbol table is the primary purpose of pass one. Pass one is necessary because the assembler does not know in advance what symbols (labels) a programmer is going to use in a given program, nor on what lines he or she will place them. Pass one reads the entire program (a

Symbolic Code	Machine Code
INP	0
OUT	1
ADD	2
SUB	3
LDA	4
STA	5
HLT	6
CMP	7
JMP	8
JG	9
JL	A (hexadecimal for 10)
JZ	B (hexadecimal for 11)

FIGURE 2.8 Opcode table.

pass) in order to determine both the labels themselves (their names) and the lines on which they occur (their addresses). After pass one has been completed, the labels themselves (in the first or label field of the four-field format) are no longer needed because the information about the labels, their names and where they occur, is available in the symbol table. Nor are the comments necessary after pass one (at which time they are included in the listing for the program; they have no real effect on either pass; the labels do have an effect on both passes.). Hence, on entrance to pass two, the assembler is primarily interested in the program as listed in Figure 2.9(a), with only the two fields, opcode and operand, present and the two tables necessary to translate the symbols of these fields into numbers: the opcode table (Fig. 2.9[b]) and the operand table (Fig. 2.9[c]).

The process of translation is one of table lookup. For example, for the instruction LDA A, the assembler will look up the code for LDA in the opcode table, find the value 4, and place it in the opcode field: 4 A. It will also look up the value of A in the symbol table, find the value 0, and place it in the operand field: 4 0. The machine code for the instruction is then 40 (hex) or 0100 0000 (binary). The code for ADD B will be 2 1, the code with which we began the article (0010 0001), and the code for STA C will be 5 2.

The code for HLT is 6, with no operand value, which we will take as 60. (Actually, the second field would hold the number of the interrupt code on some machines, e.g., 20 for INT 20 on the IBM PC, but we have ignored that issue.)

The assembler implements the translation process as it did in pass one, by setting the LC to zero, LC: 0, and reading a line of code at a time, bumping the LC as it goes along, until it has made another pass through the entire program (the second pass or pass two). When the assembler sees a storage allocation directive, for example, store constant, :5, or reserve space, RES 1, it creates the constant 05 and bumps the counter, or simply bumps the counter for RES and possibly zeroes out the locations reserved, as discussed above. When it sees an operation, a true opcode rather than a pseudo-op, it translates both the opcode and the operand field of the instruction.

a		b	
Opcodes and Operands Only		Symbolic Code	Machine Code
:05		INP	0
:06		OUT	1
RES 1		ADD	2
LDA A		SUB	3
ADD B		LDA	4
STA C		STA	5
HLT		HLT	6
END START		CMP	7
		JMP	8
		JG	9
		JL	A (hexadecimal for 10)
		JZ	B (hexadecimal for 11)

c	
Symbol Table	
A	0
B	1
C	2
START	3

FIGURE 2.9 Program (a) with opcode table (b) and operand table (c).

The process ends when the END statement is encountered. At such time, the assembler looks at the symbol in the operand field of the END directive to obtain the starting address of the program. In this case, the label is START, which has a value of 3 in the symbol table, so that the starting address of execution will be 3, START-EXECUTION-AT: 3. The result of the translation, upon end of pass two, is the code of Figure 2.10. The reader should verify that figure 2.11(a-c) symbolizes the machine code versions of the programs in Figures 1.4, 1.5, and 2.4.

```

0: 05
1: 06
2: 00
3: 40
4: 21
5: 52
6: 60
Start-location: 3

```

FIGURE 2.10 Machine code for program of Fig. 2.9(a).

a	b
0: 00	0: 00
1: 00	1: 00
2: 00	2: 00
3: 00	3: 00
4: 01	4: 40
5: 40	5: 70
6: 21	6: BC (B = hex 11, JZ; C = hex 12, address of QUIT)
7: 52	
8: 12	7: 01
9: 60	8: 21
START: 3	9: 52
	A: 12 (A = location 10 in hex)
	B: 83 (B = location 11 in hex)
	C: 60 (C = location 12 in hex)
	START: 3
c	
0: 44	
1: 25	
2: 56	
3: 60	
4: 05	
5: 06	
6: 00	
START: 0	

FIGURE 2.11 Machine codes for programs of Figs. 1.4(a), 1.5(b), and 2.4(c).

3.0 THE EXECUTION OF THE PROGRAM

The execution of the program takes place on (in) the hardware of the computer. When the program of Figure 2.10 is ready to execute, the operating system (or loader) will place the value 3 in the PC and execution will start on line 3, PC: 3. The first instruction executed is 40, which used to be LDA A. The interpretation of an instruction, such as LDA A, or 40, is as explained earlier: The data value at location 0 is to be loaded into the AC. Because location 0, which used to be A but is now referred to by number (computer address), contains a 5, the value of the AC will become a 5, that is, 0: 5 → AC: 5. The actual transfer is accomplished by sending bits (voltage levels) over an internal "bus"—a set of copper (or other conductor) connectors (one per bit) that connect the memory location to the AC (Fig. 3.1) (usually through an intermediate register).

We have listed the data values vertically, rather than horizontally, and in binary form, 101 = 5, in Figure 3.1, to indicate the role of the bus or "data path."

Memory Location	Bus	AC
0	—>	0
0	—>	0
0	—>	0
0	—>	0
0	—>	0
1	—>	1
0	—>	0
1	—>	1

FIGURE 3.1 Transfer of data values.

3.1 FETCH, BUMP, AND EXECUTE

The reader will remember that the PC contains the value (location or address) of the instruction to be executed next. Because the operating system or loader placed the value 3 in the PC for this program—the value obtained in pass two of the translation process (from the END statement)—execution began on line 3. The next instruction to be executed is determined by the PC as well. It will, in the normal course of events, be the instruction on line 4 of the program: 21, or ADD B.

The transition from the execution of the statement at location 3 to that at location 4 is accomplished by bumping (adding 1 to) the PC. This is done automatically in the hardware of the machine but parallels the bumping of the LC, which was done by the assembler (software) in the translation process of passes one and two.

The Instruction Cycle

The actual interpretation and execution of the instruction is accomplished by fetching the instruction from its main memory location, 3, for the instruction 40, 3: 40, and "moving" it to the instruction decoder (IR in Fig. 1.1), IR: 40. The decoder breaks the instruction into its two parts:

```
IC: 4
MAC: 0
```

and then uses these two parts to execute (carry out) the instruction.

The sequence in the instruction fetch and decode cycle is then

```
PC: 3  —> (3: 40 —> IR: 40)  [fetch]
IR: 40 —> IC: 4                [decode]
MAC: 0
```

The value in PC indicates the address of the instruction to fetch, 3; the fetch itself is a data transfer of the contents of location 3 to the IR. The IR is then interpreted through the predefined format for the instruction: the left half, the 4, going into the IC, and the right half, the 0, into the MAC. At this point, the machine enters the execution cycle.

The IC value, 4, is interpreted by circuitry that causes the data paths from memory to the AC to be activated, enabling a "data transfer." The exact memory location to be transferred is the location named in the MAR (machine address register) or MAC (memory address code) in our machine.

Instruction Cycles and Data Cycles

The instruction is transferred across the same internal paths (bus) as the data. Hence, when the instruction is fetched, there is a transfer of bits from the memory location 3 (main memory location) to the IR (memory location in the processor unit; Fig. 3.2). The IR then separates the two parts of the instruction, opcode and operand, according to some predetermined format. For our simple machine, the format is that the left-most 4 bits go into the IC register to indicate "what to do," and the right-most bits go into the MAC to indicate "with what."

The "with what" portion of the instruction, for a load, causes another transfer from main memory to processing unit—a data transfer. Hence, whether a set of bits obtained from the main memory is interpreted as an instruction or as data depends on whether it was transferred in the fetch portion (instruction cycle), or data portion (execution cycle) of the sequence fetch, execute.

NOTE: The circuitry that interprets the instruction is also referred to as an instruction decoder. There is one use of the term to refer to the register that receives the instruction, IR and then IC, and another to the circuitry attached to that register, location IC. The location IC may be a "logical" location, that is, the circuitry may be attached to the left-most 4 bits of the IR[1, 12–14].

Bump

The one component missing is the bumping of the PC. This is done, in our machine and in most real machines, between the fetch cycle and the execution

Main Memory Location 3		IR
0	→	0
1	→	1
0	→	0
0	→	0
0	→	0
0	→	0
0	→	0
0	→	0
0	→	0

FIGURE 3.2 Instruction transfer.

cycle of carrying out the command

```
fetch
bump
execute
```

and is illustrated in Figure 3.3.

The total process is, then, one of

```
fetch
bump
execute
```

That is, look at the PC and fetch the instruction it says to fetch, bump the PC, and interpret and carry out the command fetched. The sequence of steps necessary to carry out the command varies with the instruction. For example, the next instruction to be executed in the program of Figure 2.10, reproduced here as Figure 3.4 for ease of reference, is 21, which is the machine code for ADD B.

The execution of this instruction causes a data transfer from memory location 1 to a special operand register in the processing unit, 1: 06 → operand: 06, and the contents of this operand register and the contents of the AC to be fed into the adder,

```
operand: 06 →
AC: 05 → adder → AC: 11
```

with the result placed back in the AC. The IRs of the processing unit now have the values operand: 0 and AC: 11. The contents of the operand register end up as zero (the bits were shifted into the adder and then zeroed out in a process known as "shifting" the bits, that is, moving them, one at a time, into the adder); but the main memory location, 1 or B, still contains the 6, 1: 06.

```
PC: 3
Instruction fetched from location 3:
3: 40 → IR: 40
PC Bumped (PC ← PC + 1):
PC: 3 → PC: 4
Execution of instruction, 4 load (IC: 4)
with data value in location 0 (MAC: 0),
causes contents of location 0 to become
contents of the AC:
0: 5 → AC: 5
due to circuitry implementing
the opcode 4, circuitry "opening"
or "enabling" the data path from
main memory to the processing unit.
```

FIGURE 3.3 The bumping of the PC.


```
0: 05
1: 06
2: 00
3: 40
4: 21
5: 52
6: 60
PC: 4
```

FIGURE 3.4 Machine code for program in Fig. 2.9(a).

Explanations of adders, both "half" and "full," can be found in textbooks in computer science [2,15,16] and electrical engineering [17]. Also see the laboratory manuals for courses in computer architecture [13]. In studying the circuitry of the machines, one needs to be familiar with Boolean algebra and its relation to the construction of the circuits. A good introduction can be found in Ref. 18.

At the end of this instruction cycle, the PC has the value 5 (it was bumped), PC: 4 \rightarrow PC: 5, and the AC has the value 11, the result of the computation AC: 11. The instruction at location 5, which is 52, is fetched and executed (with a bump in between). The execution causes a data transfer from the AC to the main memory location, in this case, location 2, AC: 11 \rightarrow 2: 11. It is in this way that the location originally called C, C RES 1, is set, that is, obtains a meaningful data value.

At the end of this instruction cycle, the PC has the value 6, which causes the HLT instruction to be fetched, 60. As indicated earlier, the execution of this instruction is really a call to an operating system routine, which routine notes that this program has terminated and that another program can now be loaded or started (if already loaded). On mainframe computers this new program might be another user program. On a PC the new program is usually the command interpreter of the operating system, that is, DOS. (DOS stands for disk operating system and comes in two popular forms: MS-DOS for non-IBM micros, some of which emulate [or clone] the IBM PCs and some of which do not; and PC-DOS, for the IBM products).

3.2 EXECUTION OF THE PROGRAM WITH INPUT AND OUTPUT

The execution of the program with input and output is essentially the same as that just explained with the exception that there would be other calls to the operating system. The machine code and starting address (value in PC) are shown in Figure 3.5.

Execution begins on line 3. The instruction is fetched, 3: 00 \rightarrow IR: 00, and decoded:

```
IC: 0
MAC: 0
```

The value in the IC causes a call to the operating system to read from the keyboard. If the person at the keyboard types a 5, this value will be placed

```
0: 00
1: 00
2: 00
3: 00
4: 01
5: 40
6: 21
7: 52
8: 12
9: 60
PC: 3
```

FIGURE 3.5 Machine code and starting address.

in memory location zero (MAC: 0), keyboard: 5 \rightarrow 0: 05. After the call (interrupt to the operating system), control will be transferred back (returned) to this program, the user program, and the next instruction will be executed. This is the instruction at location 4, 01, which causes a read (IC: 0), with the data value placed into the memory location 1 (MAC: 1). At the end of the execution of these two instructions, the memory looks like that of Figure 3.6. The data values in A and B (locations 1 and 2) have been initialized (set) to 5 and 6, respectively (what the person typed in); and the PC has been bumped to 5, the address of the next instruction to be fetched and executed, 40, for LDA A.

The operation of the next few instructions is the same as the preceding program. However, in this program, after the value of C is set to 11, a call to the operating system, 12, is executed. The call instructs the operating system to output a data value to the screen (IC: 1). The data value output is to be obtained from location 2 (MAC: 2). This effectively prints the value in location C, 11, on the screen.

```
0: 05
1: 06
2: 00
3: 00
4: 01
5: 40
6: 21
7: 52
8: 12
9: 60
PC: 5
```

FIGURE 3.6 Memory after instruction execution.

Instructions or Data

The program of Figure 3.5 is interesting if one compares the initial contents of memory in locations 0 and 3:

```
0: 00
3: 00
```

The contents are the same. There is no essential distinction between the contents of a data location and the contents of a main memory location at the machine level. If the contents of location 0 were fetched in the instruction cycle, PC: 0 \rightarrow (0: 00 \rightarrow IR: 00), the data would be interpreted as a read instruction, the result of the read to be put into location 0!

One can tell what is data and what is instruction at the machine level (and, in a sense at the assembly code level) only by keeping the intended data and the intended instructions separate, physically, and arranging for the IC to begin with the address of an intended instruction (by properly constructing the end statement).

In programs with bugs in them, a value of an intended data location sometimes makes its way into the PC. The result is either (a) a message that the instruction is illegal, if the opcode (value in IC) is not a legal opcode; (b) a memory address is out of range, if the data value in the MAC is not an address in the area of memory allocated to the program; or (c) in an instruction (unwanted), that is, an add, load, store, or other instruction being executed, if the contents of the IC and MAC are both valid values. In the third case, because the execution of the instruction was not intended, the results are sometimes surprising. "Unpredictable" is the term usually used, but the results are exactly predictable, if one knows what location was unintentionally taken to be an instruction. The contents of the location tell one how the bit pattern was interpreted, like it or not, that is, interpreted "by accident."

Because of the unpredictability of the results of logical errors in an assembler program, the debugging of ASM programs is sometimes fascinating (sometimes frustrating) and usually involves one in examining the numeric values in the various memory locations, both the main memory values and the values in the registers of the processing unit (AC and PC). The examination of these values is the purpose of programs that "dump core" (dumps), or that allow one to step through a program (debuggers). (The phrase dump core is still used to refer to the printing of the values in the storage locations of main memory and the processor, although few machines still use core memory.)

Saving the PC and the AC

When a call is made to the operating system, it too must use the computer. Like our programs, it uses the main memory locations allocated to it (for its instructions and data) and the registers of the processing unit (PC, AC, etc.). Some of these registers (PC, AC) contain data values of interest to our program (the next thing to be done [PC] and the current results [AC]). These values are saved

```
PC: 9  --> Saved-PC: 9
AC: 11 --> Saved-AC: 11
```

before or upon transferring control to the OS, and reloaded upon return to the user program:

```
Saved-PC: 9  → PC: 9
Saved-AC: 11 → AC: 11
```

Our program can then "take up where it left off." The saving and initializing of the registers is done either by the user program, operating system software or, in some machines (e.g., the DEC/VAX) by specialized hardware that is activated upon the occurrence of the interrupts.

NOTE: The VAX hardware includes two special commands to facilitate the switching of "contexts," that is, user program running to operating system running and vice versa. They are SVPCTX for "save context," the context being the contents of the PC, AC, and other "interesting" registers; and, on the return, LDPCTX for "load context," which restores (reloads) the saved values, before turning execution over to the user program (by loading the PC). Explanations of SVPCTX and LDPCTX can be found in the DEC hardware reference manual and in Ref. 19.

3.3 EXECUTION OF THE JUMP AND COMPARE INSTRUCTIONS

The execution of the program with jump and compare instructions brings up a new topic, the transfer of control. The topic is best explained with regard to the unconditional jump, JMP BEGIN, which is shown in coded form on line B (hex for 11) of the machine code for the program (Fig. 3.7).

The state of the machine, just before the fetching of the jump instruction on line B, is as shown in Figure 3.8(a,b). The values of locations 0, 1, and 2 have been set by the two reads, 00 on line 3 and 01 on line 7, and the store, 52 on line 9, in Figure 3.8(a). We have also seen the result of the computation, 11, on the screen, due to the execution of the output statement, 12 on line A (hex for 10). The primary locations of interest, if we dumped core, would be the PC, AC, and the three storage locations, A, B, and C (Fig. 3.8[b]).

```
0: 00
1: 00
2: 00
3: 00
4: 40
5: 70
6: BC  (B = 11, JZ; C = 12, the value of QUIT, i.e., its address)
7: 01
8: 21
9: 52
A: 12  (A = address 10 in hex)
B: 83  (8 = JMP; 3 = BEGIN, the label that was on line 3)
C: 60  (C = hex for 12, address of prior label QUIT)
```

FIGURE 3.7 Machine code for unconditional jump.

a

```

0: 05
1: 06
2: 0B (hex for 11, the result of 5 + 6)
3: 00
4: 40
5: 70
6: BC
7: 01
8: 21
9: 52
A: 12
B: 83
C: 60
PC: B

```

b

```

PC: B (address of JMP instruction 83)
AC: B (result of adding 5 and 6)
A: 5 (result of INP A)
B: 6 (result of INP B)
C: B (result of STA C)

```

FIGURE 3.8 (a) Storage locations for program in Fig. 3.7 after execution of lines 3-A; (b) dump of locations of interest for a.

The jump instruction, 83, is fetched from location B (location 11), PC: B \rightarrow (B: 83 \rightarrow IR: 83), and decoded:

```

IC: 8
MAC: 3

```

The instruction part (IC: 8) indicates that control is to be transferred to the location in the operand field (MAC: 3). This is accomplished by placing this value (3) in the PC, which currently holds the value C, PC: C, because it was bumped in the fetch, bump, and execute cycle for the JMP instruction:

```

Fetch cycle, PC: B
Bump, PC: B + 1  $\rightarrow$  C
Value of PC during execution of line B--PC: C

```

Hence, if nothing else happened, the instruction on line C, 60 for HLT, would have been executed next. But now, during the execution cycle, the JMP (opcode 8) causes a data transfer from the MAC to the PC, MAC: 3 \rightarrow PC: 3. Thus, upon conclusion of the jump instruction, the PC has the value 3, PC: 3, and that is the location of the instruction to be fetched and executed next.

```

BEGIN: INP A
      .
      .
      .
      JMP BEGIN

```

FIGURE 3.9 Assembly language code.

Jumps are always accomplished via a change in the value of the PC. The value in the operand field, which is an address (now held in the MAC), is placed into the PC. It is the address of the instruction to be executed next and is the value of the label that was placed on that line (the object or target of the jump) in the original assembly language code (Fig. 3.9).

Conditional Jumps

The only distinction between the unconditional jump (JMP) and the conditional jumps (JL, JG, JZ) is in whether or not the jump is actually taken (the value of MAC actually transferred to PC). When the condition being tested for is fulfilled (the zero flag is set, i.e., hardware 1, for "yes, zero," in the case of JZ), the jump is taken. When the condition is not fulfilled (zero flag was 0, "not zero," as opposed to 1, "yes zero," for JZ), the jump is not taken.

The execution of the conditional jump, IC: B (from IR: BC), involves the checking of the flags (P, N, Z), in the case of JZ, the Z flag, to see if it is set (to 1). If it is, the value in the MAC

MAC: C (C = hex 12, the address originally labeled QUIT)

is placed in the PC, MAC: C \rightarrow PC: C, which causes the transfer of control. If it is not, the PC is left at its current value, which is the instruction following the jump, PC: 7, because it has been bumped from PC = 6 after the fetch of the conditional jump instruction:

```

PC: 6
6: BC  $\rightarrow$  IR: BC
PC  $\leftarrow$  PC + 1
PC: 7

```

The compare (CMP)

The compare, as depicted for ENC

```

IC: 7
MAC: 0 (not necessary)

```

causes the examination of the value in the AC and the setting of the flags. If the value is positive, indicated by a zero in the first (left-most or high order) bit,

```

0 0000101
|
sign bit

```

The P bit is set (to 1). If the left-most bit is a 1, 1 _____, the value is negative, and the N bit is set (to 1). Finally, if the magnitude of the instruction, all bits except the sign bit, is zero,

```

0      000 0000
sign   magnitude

```

the Z flag (Z bit) is set.

NOTE: We have not shown the actual bits for the negative number, sign bit 1. They would be 11111010 in one's complement and 11111011 in two's complement. The representation of numeric values in the computer, which usually includes the representation of negative numbers in one's complement or two's complement, as well as the representation of floating-point numbers, is discussed in the standard texts [2,20].

Subroutines

The call to subroutines in assembly language is essentially the same as that of a jump instruction, with the exception that the value of the PC (the instruction to be executed upon return from the subroutine, is saved before jumping (transferring control to the subroutine).

A program that illustrates the use of a subroutine using the structure of our machine is that of Figure 3.10.

Before going to the subroutine (CAL ADDNUM), the value of the PC (7; it has already been bumped) plus 1, $7 + 1 = 8$, is saved in the location SAVEPC. This is accomplished by the instruction on line 6: SPC SAVEPC. The opcode SPC causes the contents of the PC, to be loaded into one of the processor registers, to be bumped by one (to account for the CAL instruction) and to be stored in the location named in the operand field:

PC: 7 \rightarrow special register that adds 1 to the value: $7 + 1 \rightarrow 8$
 and stores this result in the location named in the operand field: 8 \rightarrow SAVEPC: 8

```

0: A:          RES 1
1: B:          RES 1
2: C:          RES 1
3: SAVEPC:     RES 1
4: EX:         INP A
5:             INP B
6:             SPC SAVEPC
7:             CAL ADDNUM
8:             OUT C
9:             HLT
A: ADDNUM:     LDA A
B:             ADD B
C:             STA C
D:             RET SAVEPC
D:             END EX

```

FIGURE 3.10 Example of a subroutine.

The value of the PC itself, 7, is unchanged.

NOTE: The "plus one" in saving the value of the PC is done to account for the intervening call instruction, CAL ADDNUM; we want control to return to the instruction after the jump to the subroutine. This convention, that SPC causes the value of the PC plus 1 to be saved, would be set out ahead of time by the manufacturers of the computer and indicated in the hardware reference manual as SPC: (PC) + 1 → (MEM). The end result is that the memory location named in the operand field, SAVEPC, has the "return address," the address of the instruction to be executed upon return from the subroutine, SAVEPC: 8, which is the instruction to output a data value to the screen:

```
8: OUT C (1C: 1 = output to screen
           C = contents of location C)
```

The transfer of control to subroutines and the subsequent return are handled differently on different machines. However, the essence of the procedure is to (a) save the address of the instruction to be executed upon return and (b) return "through" the location saved.

The call to the subroutine, line 7, which is executed next (the PC still has 7 in it; its value was not changed), CAL ADDNUM, then transfers control to the subroutine. The mechanism of transferring control is the same as it is for the jump instruction. The address named in the operand field, ADDNUM = A (hex 10), is currently held in the MAC, MAC: A. This value is placed into the PC, MAC: A → PC: A, so that the "next thing" executed is the instruction on line number A, the line labeled ADDNUM, line number A = ADDNUM: LDA A. On return from the subroutine, RET SAVEPC, the address in the place named in the operand field is placed into the PC, SAVEPC: 8 → PC: 8, so that the next thing done is the instruction on line 8, 8: OUT C, the instruction after the call.

Although the actual operation of subroutines is slightly more complex than that shown here, the basic idea is the same: (a) Save the value of the PC; (b) transfer control to the subroutine; and (c) return from the subroutine through the value of the saved PC.

In the subroutine shown in Figure 3.10, no arguments were passed. It was assumed that the input data values would be in locations A and B and the output would be placed in C. In real assemblers there is more flexibility built into the passing of arguments. One usually designates the locations to be used in the subroutine by placing their addresses after the call (Fig. 3.11[a]), or on a "stack" used to pass data values (Fig. 3.11[b]). However, whatever the convention, the procedure is the same: The locations, or the data values of the input variables (A, B), are transferred to the subroutine; it manipulates them and returns a result (C).

NOTE: If the subroutine uses any registers (ACs) the values in these registers must be saved upon entry to the subroutine and restored before exit. The procedure is the same as on calls to the operating system, except it is the responsibility of the subroutine (person coding the subroutine) or the calling routine (person writing the "main


```

a
CAL ADDNUM
A
B
C
END-LIST

```

```

b

```

```

A
B
C

```

FIGURE 3.11 (a) Argument list in ASM code; (b) arguments on stack.

line" code) to do the saving and restoring (write the code that saves and restores the values. Just whose responsibility it is varies from system to system).

Macros

A macro is similar to a subroutine except that it involves no transfer of control. An example is shown in Figure 3.12.

The macro is defined in the portion of code called DEFM: DEFM ADDNUM. The name of the macro is ADDNUM, and its form is ADDNUM \$1 \$2 \$3. The numbers \$1, \$2, and \$3 stand for argument "places" and will be replaced by the actual variable names, that is, the location names (A, B, and C, in this case) when the macro is "invoked" (called) by the line ADDNUM A B C, called the reference to the macro.

The code in the definition of the macro indicates that the value of the first argument in the reference will be loaded as LDA \$1, the value of the second argument added to it as ADD \$2, and the result stored in the third

```

DEFM ADDNUM
ADDNUM $1 $2 $3
LDA $1
ADD $2
STA $3
END-MACRO-DEF
A: Res 1
B: Res 1
C: Res 1
EX: INP A
    INP B
    ADDNUM A B C
    OUT C
    HLT
    END EX

```

FIGURE 3.12 Example of a macro.

argument as STA \$3. The end of the macro definition is indicated by the pseudo-op END-MACRO-DEF.

The actual invocation of the macro takes place at the call, or reference to the macro: ADDNUM A B C. The invocation causes the lines of code defined in the definition to be placed, physically, in this and the following locations, replacing the argument forms (placeholders), \$1 \$2 \$3, by the names occurring in the actual call, A B C. Hence, the code

```
LDA A
ADD B
STA C
```

is generated, and placed, physically, immediately after the statement INP B:

```
INP B
LDA A
ADD B
STA C
```

Because the code is placed physically "in line," the macro is referred to as an "in-line subroutine" or as an "open subroutine" (subroutines with real calls, i.e., transfers of control, are called closed subroutines).

There is no transfer of control in using a macro. The code is generated "in place." Its use is a convenience. It can shorten programs that use the same set of code in several places, for example, several invocations of ADDNUM, and it makes the code more "readable": ADDNUM A B C. The shortening of the code is in appearance only, that is, in the listing of the program. At run time the code is as long as it would be if one had written it out. It does, however, run somewhat faster than code using subroutines because there are no transfers of control and, hence, no saving and transferring of data values.

Not all assemblers provide for macros. Assemblers that do are referred to as macro assemblers. Good treatments of assemblers, both macro and non-macro, as well as other systems software, can be found in Refs. 21 and 22.

3.4 SUMMARY OF THE ASSEMBLY AND EXECUTION PROCESS

In this section we have shown how an assembler works: (a) Pass one creates a symbol table, which is a list of the labels and the lines (addresses) on which they appear; (b) pass two uses the symbol table, generated in pass one, and the opcode table, provided by the manufacturer, to translate the assembly language program.

The result of the translation is machine code, which is executed on the hardware of the computer, by fetching instructions, decoding them, and carrying them out. In carrying them out, data sometimes have to be fetched as well (data cycle). The instruction to be fetched (instruction cycle) is indicated by the contents of the PC, which is bumped, usually between the fetch and the execution, on every "machine cycle." The data to be fetched are indicated by the contents of the MAC, and the operation to be performed is indicated by the contents of the IC.

Transfers of control, for example, jumps, are accomplished by simply changing the value of the PC from what would have been the next instruction

executed to the value (address) in the operand field. If a return is anticipated, as in the execution of a subroutine, the return address must be saved. Any data values that may be altered by the subroutine, for example, the value of the accumulator, should be saved by the subroutine or main program on entry and restored before return. Macros are like subroutines except that they generate code "in place." There is no transfer of control and no saving of data values.

The execution of the assembly language (machine language) program sometimes causes an interaction with the operating system, for example, in calls to read and write or to halt. These calls, usually implemented as "interrupts to" the operating system, act like subroutines, with the operating system routine playing the role of the subroutine, the user program, the calling routine.

NOTE: At the machine code level, one cannot tell if code is the result of an assembly language program or a higher level language program (except possibly through its form, that is, the type and sequence of machine statements used). The code is all "of a piece" at the machine level, having been generated by an assembler if the original code (the "source" code) was assembler, or by a compiler if it was a higher level language, such as Pascal. An interpreter, such as Basic, works in a slightly different mode, translating and executing the code a line at a time, so that the analogy between assemblers and compilers does not strictly apply to them. Assemblers and compilers translate the entire program, "all lines," before executing it, by making one or more passes over the program.

4.0 OTHER ASSEMBLERS

The assembler shown here is referred to as a "one-address" assembler and the machine, a one-address machine. The phrase refers to the fact that there is only one address in the operand field (or none): LDA A. (We should say "at most one" because some instructions, such as HLT, might take zero operands.)

In a one-address assembler, source or destination address is taken for granted (is implicit in the instruction). This is the AC in a one-address machine. For example, in the instruction LDA A, the source field is the memory location A; the implicit destination of the instruction is the AC. In the instruction STA C, the source is the AC, implicit, and the destination is C, explicit.

Instructions that take two input variables, such as ADD B, which is a binary operation, ADD X,Y, have one of the input variables implicit, the AC, ADD AC,B, and the destination, the location that receives the result, also implicit, again the AC ADD AC,B,Result to AC.

Two-Address Machines

Single-address machines are possible when one uses a single AC. Its value is always implicit; however, the use of a single AC can become cumbersome. Hence, most machines have more than one AC, in which case they are referred to as registers or arithmetic registers.

NOTE: There are various types of registers: arithmetic registers, base registers, index registers, and so forth, each type serving a distinct purpose. For example, the UNIVAC 1108 had 48 registers: 16 arithmetic (called arithmetic or "A registers"), 16 index (called "X registers"), and 16 for transfer of control ("R registers"). The arithmetic was done in the A registers, numbered A0 through A15, whereas the other registers were special purpose, the X registers being used for indexing (see below) and the R registers for counting off loops (also see below). The IBM 360 had 16 general-purpose registers, that is, they were used for both arithmetic and addressing. The IBM PC, which utilizes the 8088 (XT) or 80286 (AT) microprocessors [23], has 12 registers besides the PC, which is called the IP (instruction pointer) on the PC. (PC = computer, not program counter.) The PC also has eight flags (overflow, interrupt, sign, zero, and carry, among others).

When one has more than one register, one must not only designate the memory location to be used in the instruction but also the register (AC) to be used. Hence, one writes two-address operands:

```
LDA  A1,A
ADD  A1,B
STA  A1,C (UNIVAC 1108 type instructions)
```

The instructions indicate that the memory location A is to have its value loaded into register A1; B is to have its value added to the same register. (Why? We are trying to add B to A, and that is the location of A); the result is to be stored in C.

Most current machines are two address. The PDP-11, a minicomputer, has eight registers, numbered R0 through R7. R0 through R5 are used as accumulators, whereas R6 is used as a stack pointer (SP) and R7 is used as the PC. The VAX (PDP VAX 11/750, 780, 785, etc.), a super minicomputer, has 16 registers: 12 for general use (R0 through R11) and 4 for special purposes, including the SP and the PC. As indicated above, the IBM PC, which uses either an 8088 or an 80286 microprocessor (manufactured by Intel, Inc.) has 12 registers besides the PC or IP. Some of these (the segment registers), for example, DS (data segment register) and CS (code segment register), are used for addressing (see below). Others (A, B, C, and D) have more general uses. (The DS and CS registers can be used to separate the code [instructions] from the data, as alluded to earlier.)

The primary arithmetic register for the PC is the A register. One refers to the A register on the PC as AX, AH, or AL. The PC registers are 16 bits long (16-bit word size, 2 bytes per word). One can, however, refer either to the entire 16 bits or to either half of the word. If one refers to the entire 16 bits, one uses the designation AX: MOV AX,A (load AX with the value of A). If one refers to either half, one uses AH (for "high") or AL (for "low"):

```
MOV AH,A (load high half of AX with A)
MOV AL,A (load low half of AX with A)
```

The words high and low refer to the most significant bits (high order) and least significant bits (low order). The word has 16 bits, numbered from 0 through 15, right to left:

AX
15 14 13 12 ... 0

so that the high-order half-word AH consists of bits 8 through 15, and the low order half-word AL consists of bits 0 through 7. When referencing the register, the source (sending) field should be a word (16 bits long) when referencing AX and a byte (8 bits long) when referencing AH or AL.

Code to load A, add B, and store the result in C would look like the following for the PC:

```
MOV AX,A
ADD AX,B
MOV C,AX
```

The MOV instruction causes both loads and stores, the position of the operands indicating what is a load and what is a store.

Interpreting the Operands

In the UNIVAC-type instructions,

```
LDA A1,A
ADD A1,B
STA A1,C
```

the placement of the AC was fixed. It came first, and the memory location came second. What was considered a source field and what was considered a destination depended on the instruction. In the load instruction, the memory was the source, and the AC was the destination, LDA: (MEM) \rightarrow (AC). In the store it was the reverse, STA: (AC) \rightarrow (MEM). The form of the instructions, with respect to the placement of the AC field, was static:

```
LDA A1,A (load)
STA A1,A (store)
```

In the instructions for the IBM PC, the position of the AC is not fixed; however, the concept of source and destination is fixed. The format is MOV des+, source, or MOV to, from. Hence in the instructions MOV AX,A, the source is A, and it is "going to" AX. In the instruction MOV C,AX, the source is AX, and it is moving to C. The instruction MOV AX,A is then a load, whereas MOV C,AX is a store.

Whether one keeps the location of the AC field fixed (in the operand field of the assembler instruction), and determines the source and destination fields from the instruction (UNIVAC, DEC-10, IBM 360/370) or one keeps the source/destination positions fixed and moves the AC and memory names accordingly (PDP-11, VAX, IBM PC), is machine dependent. It is immaterial, as long as one knows what convention is being implemented. (One should also determine, in the source/destination convention, what is the source and what is the destination. For example, for the VAX, the sequence is mov source, destination, or mov from, to, so that the code,

movl a,r1 (movl = move long, long = a word, 32 bits on the VAX)

is a load, whereas `movl r1,a` is a store. The scheme is reversed on the IBM PC: `mov to, from`, rather than `mov from, to`. The variety of formats keeps one on one's toes.

Three-Address and Four-Address Machines

Earlier machines used three- and four-address instructions [11,20,21,22]. The four-address instructions, `ADD X1, X2, X3, NI`, indicated both the input variables (`X1, X2`) and the name of the variable in which the result was to be placed (`X3`), as well as the address of the instruction to be executed next (`NI`). The execution of the instructions did not have to be sequential.

The fourth address (`NI`) was eliminated by keeping the address of the next instruction in one location, the `PC`, and assuming sequential execution (bump `PC` by 1) unless told otherwise (a transfer of control). The instructions still had three fields, `ADD X1, X2, X3`, so that the result of the computation could be placed anywhere (`X3`).

The three-address machines became two-address machines by assuming that one of the variables, for example, the `AC`, was both a source (input) variable and a destination (output or result) variable in a computation: `ADD A1,B`. This is the format that is popular today.

NOTE: There are also zero-address or stack machines, such as calculators [11].

4.1 FIXED-LENGTH VERSUS VARIABLE-LENGTH INSTRUCTIONS

The machine instructions shown here have been fixed length, each 8 bits long: 4 bits for the opcode and 4 bits for the operand code. Fixed-length instructions have been used on some machines, such as the UNIVAC 1108 and the DEC-10 (both mainframes with 36-bit word size and 36-bit instruction formats); other machines, such as the IBM 360 and its descendants (E.G., 370, 43XX, 303X, 308X) and the PDP series (e.g., PDP-70 and PDP-11/VAX) have used variable-length instructions.

The IBM PC is, for the most part, a fixed-length instruction machine. The instructions are, with the exception of constants and addresses, 2 bytes long. The first byte indicates the operation to be performed and the register to be used in the operation. The second byte indicates the memory location to be used. It is either a data value or the address of a data value. For example, in the command `MOV AH,1`, the data value 1 is to be moved into the register `AH` (to prepare for inputting a character from the keyboard: `INT 21`, with `AH = 1`). The information that the data are to be moved into `AH` is included in the opcode for the instruction `B4`, which indicates a move involving `AH`. The data to be moved, the 1, is included in the operand field, which will consume 1 byte for the value 1: `B4 01`. The entire instruction is then 2 bytes long. There is some variation in the instruction length, for example, if a larger constant is used, say `1234h` (`h = hex`): `MOV AX,1234h`. The entire instruction, opcode plus constant, will be 3 bytes long, `B8 1234`, but the distinction is primarily one of moving bytes to bytes (1 to `AH`) or words to words (`1234h` to `AX`). This distinction is indicated in the opcode, `B4` for move byte, `B8` for move word (when the `A` register is involved). (The value `1234` is stored backwards in the `PC`, `B8 3412`, but we omitted that detail for clarity [3]. Although the distinction between moving bytes and words does result in

variable-length instructions, it would probably be stretching the point to consider the microprocessor to have a variable-length instruction set. However, it does face the same problem as machines with variable-length instruction sets: how to bump the PC?

In machines with a fixed-length instruction set, such as ENC, the bumping of the PC is uniform: one word each time, $PC \leftarrow PC + 1$. In machines with variable-length instruction sets, the PC must be bumped according to the length of the instruction. On the microprocessor, this would be 2 bytes for the byte instructions and 3 bytes for the word instructions (involving constants).

Machines with variable-length instruction sets are "byte addressable," rather than "word addressable."

NOTE: A word consists of 2 to several bytes, such as 2 for a 16-bit word and 4 for a 32-bit word. ENC was an anomaly, with the word size (8 bits) equaling the byte size. The early microprocessors were of these smaller sizes, for example, with 4-bit and 8-bit processors [see Refs. 24 and 25; other texts specifically about the architecture of microprocessors and microcomputer systems are Refs. 26 and 27].

In a byte-addressable machine, each byte is labeled (Fig. 4.1). One can then bump the counter by any number of bytes: 2 for the 2-byte instructions, $IP \leftarrow IP + 2$, and 3 for 3-byte instructions, $IP \leftarrow IP + 3$. As indicated above, both the IBM 360/370 series and the PDP series of machines were byte addressable, with variable-length instruction sets [6, 28-30]. The IBM 360/370 series had instructions that were 2 bytes long (half-word), 4 bytes long (full word), and 6 bytes long (one-and-a-half words). The PC was then bumped accordingly, by 2, 4, or 6, depending on the instruction involved. Because this occurs in the circuitry of the machine, it is of no concern to the programmer, until he or she has to debug a program and interpret the contents of the PC. (PC = program counter not microcomputer; the acronyms are difficult to keep track of.)

In machines with variable-length instruction sets, the length of the instruction is determined by the opcode and the mode of addressing. We have, for the most part, assumed one mode of addressing: "direct addressing." This simplification (necessary, and not oversimplification) is rectified next.

4 bytes per word	2 bytes per word
0 1 2 3	0 1
4 5 6 7	2 3
8 9 A B	4 5
.	6 7
.	8 9
.	A B
	.
	.
	.

FIGURE 4.1 Byte-addressable system.

4.2 MODES OF ADDRESSING

The term addressing is used to designate the method of determining the address of the operand to be used in a particular instruction. This address, that is, the final address to be used in carrying out the instruction, is referred to as the "effective address." It is not, as has been shown in the examples above, always the address we start out with.

There are several modes of addressing, some of the more common being direct, indirect, immediate, base displacement, and indexing. The programs shown here have employed direct addressing (for the most part; the RET SAVEPC was a type of indirect addressing, although it was indirect "in the hardware" as was explained above).

Direct Addressing

In direct addressing, the contents of the location named in the operand are moved (or added, etc.) to the AC, LDA A: (MEM) \rightarrow (AC). The effective address is A, and the value of A (its contents) is what is loaded.

Indirect Addressing

In indirect addressing, the address named in the operand field is not the effective address. Rather, its contents are taken to be an address, and the "contents of the contents" of the location named in the operand field are loaded. For example, if the locations A and B have the values A: 5 and B: A, the "indirect" command

```
LDA *B  (load "what's at" B, the * indicating "what's at" or indirect
        addressing)
```

would cause the contents of B to be interpreted as an address, B: A \rightarrow Address: A, and the data to be loaded from that address, A. Hence, the value of the AC becomes 5. The asterisk in the command LDA *B, is the indication that the value in B is to be taken as an address rather than as a final data value.

The Effective Address

The effective address is the "final address" used in carrying out the instruction. Hence, in the instruction involving direct addressing, LDA A, the effective address is A. In the instruction LDA *B, the effective address is again A (the contents of B), rather than B. B is an "intermediate address," whose value is used in the process of obtaining the final or effective address.

Returning from a Subroutine

One common use of indirect addressing is the return from a subroutine. If the return address is saved in a given location, say SAVEPC, SAVEPC: 8, one would return by the command RET *SAVEPC, which has the effect of the command JMP *SAVEPC. The jump would be made, not to the location called SAVEPC but to the location contained in SAVEPC, which is 8 in this case. The address 8 is the effective address, not the address SAVEPC (which was 3, line 3, in the program of Fig. 3.10).

Immediate Addressing

In immediate addressing, the data value to be loaded, for example, 2, is given in the operand field, rather than the address of a location holding the data value. An example is the command `LDA [2]`, which places the value 2 in the AC. The brackets indicate the intention to have immediate addressing. The construction `[2]` is referred to as a literal. It acts as a constant in a higher level language. The example of the IBM PC, `MOV AH,1 (B4 01)`, was an example of immediate addressing, although the PC did not use the convention of enclosing the literal (constant) in brackets.

Base Displacement Addressing

In base displacement addressing, the value in one location, a register (the base value), is added to the value in another location, the operand named in the command, to obtain the final value. For example, if the register BR is the base register, the value in it, `BR: 0000F000`, would be added to the address A in the command `LDA A(BR)` to obtain the final address. If A = 7 in the symbol table, the calculation would be

```
(BR) = 0000F000
  A  =      7
EA  =    F007
```

The base displacement method of addressing was popular with IBM machines, both mainframes—the 360 and descendants—and the PC. The parentheses indicate the intention to have base displacement addressing performed, although parentheses were not needed on the 360 series, the base register being indicated in a pseudo-op: "using 15" for using base register 15.

NOTE: The displacement is calculated in a slightly different manner on the IBM PC, after a "shift left" of one hexadecimal digit (half a byte), but the concept is the same; the calculation was performed as shown on the 360.

The intent of base displacement addressing was to allow the program to reference a larger memory space than was possible in the bits set aside for the address of the operand. For example, with a 1-byte operand field, one can address only 256 locations (Fig. 4.2). However, if one adds this displacement (also referred to as an offset) to a 16-bit value stored in a base register, for example, hex FF00 (4 hex digits = 16 bits), one can effectively address 65,536 locations (Fig. 4.3).

```
00 (hex 00)
01
02
.
.
.
FF (hex for decimal 255)
```

FIGURE 4.2 One-byte operand field.

FF00 (base register value)
 +FF (offset or displacement to base register value)

 FFFF Final Address
 (FFFF = hex for 65,535, i.e., $2^{16}-1$)

FIGURE 4.3 Sixteen-bit operand field.

Indexing

Indexing operates exactly like base displacement, although it was developed with a different purpose in mind. The value in one register, the index register, is added to the address in the instruction LDA A(X1). If the index register X1 has the value 1000, and A is again location 7, the effective address would be 1007:

(X1) = 1000
 A = 7
 EA = 1007

The parentheses indicate the intention to have indexing performed.

The intent of indexing was to facilitate stepping through an array. The array would be defined with the RES statement, A RES 10, and then filled by performing a loop (Fig. 4.4). The value of the index register, X1, is initially set to zero, X1: 0. This value is added to the address of A on the first iteration of the loop (Fig. 4.5).

```

                                MOV A1, [10]      ; count for loop
                                MOV X1, [0]       ; initialize index register
GETDATA: INP A(X1)              ; input data into A indexed
                                ; by value in X1.
                                INC X1            ; add one to X1 (INC =
                                ; increment by 1, the "step"
                                ; in stepping through the
                                ; array).
                                DEC A1            ; decrement A1 by 1, decrementing
                                ; the "count" in counting the
                                ; loops.
                                JG A1, GETDATA    ; while A1's value is greater
                                ; than zero, go back to GET DATA.
                                .
                                .
                                .
                                Rest of program
                                .
                                .
                                .
  
```

FIGURE 4.4 Stepping through an array.

$$A(X1) = A + (X1) = A + 0 = A$$

|
() = contents of X1, currently 0

FIGURE 4.5 First iteration of a loop.

This effectively puts the first value input into A itself, say a 5:

INP A(X1) \rightarrow INP A + 0 \rightarrow INP A \rightarrow A: 5

The value of X1 is incremented (INC), that is, bumped by 1 (or the equivalent appropriate value on byte-addressable machines):

INC = (X1 \leftarrow X1 + 1) \rightarrow (X1: 0 \rightarrow X1: 1)

and this value is added to the address of A on the next iteration of the loop:

A(X1) = A + (X1)
= A + 1

This places the second data value in the location A + 1:

A: 5
: 6 \leftarrow A + 1

The procedure continues for 10 iterations (controlled by the value in register A1, the count register), with the last data value being placed in A + 9 (Fig. 4.6). Ten locations, A + 0 through A + 9, have been initialized and can now be processed by the rest of the program.

The loop is controlled by the value in A1. It was initialized to 10, A1: 10, and decremented each time through the loop (Fig. 4.7). The jump instruction JG A1, GETDATA says to GOTO GETDATA as long as the value in A1 is greater than zero (JG). This has the effect of "counting off" 10 loops, 9 8 7 6 5 4 3 2 1 0. On the last iteration, the value in A1, A1: 0, causes the JG condition to be false: The value in A1 is not greater than zero, the jump is not taken, and control passes to the next sequential instruction, the "rest of the program" (Fig. 4.4).

A: 5 A + 0
: 6 A + 1
:
:
:
: 9 A + 8
: 4 A + 9

FIGURE 4.6 Loop initialized to 10.

```

DEC A1: 10 → A1: 9
DEC A1: 9 → A1: 8

```

FIGURE 4.7 Decrementing value in a loop.

The Distinction between Base Displacement and Indexing

Indexing was used to step through arrays. The purpose of base displacement addressing was to create program segments. Each segment had a base address, which was loaded into the base register when that segment was wanted. All addresses within this segment were then considered as displacements (offsets) from this base segment address. For example, if a program had two segments, one and two, at locations 1000 and 2000, respectively (Fig. 4.8), one could access (choose) segment one by loading its address into the base register

```
LDADR BR, ONE → BR: 1000
```

and then reference location A by the command LDA A(BR), which causes the relative address of A, zero in segment one (its offset from the location labeled "One"), to be added to the value 1000, $1000 + 0 \rightarrow 1000$, yielding an effective address of 1000.

When using base addressing, the assembler translates each segment as if it were a program in itself, with the first statement numbered 0, the next, 1, and so forth. This is shown in Figure 4.9 for the segments of Figure 4.8. The line numbers in each segment, their addresses relative to the start of the segment, are the offsets. At execution time, these offsets are added to the contents of the base register. Hence, the relative address of A in segment one is zero, which is added to the value in the base register, 1000, to obtain the final or effective address, 1000. The address loaded into the base register is the machine address (absolute address; see below) of the line labeled "One," that is, the first line of the segment. In accessing location B by the command ADD B(BR), the effective address of B, computed by adding the value of the base register (1000) to the offset (relative address within the segment) of B (1) is 1001. These calculations are shown in Figure 4.10. One accesses segment one by loading the base register, BR, with the value 1000, the actual address of segment one (provided to the program by the loader

```

One:  A: RES 1   Line 1000
      B: RES 1   1001
      .
      .
      .

Two:  C: RES 1   Line 2000
      D: RES 1   2001
      .
      .
      .

```

FIGURE 4.8 Program with two segments.

```

One:  A: RES 1   Line 1000  line 0
      B: RES 1           1001  line 1
      .
      .
      .
Two:  C: RES 1   Line 2000  line 0
      D: RES 1           2001  line 1
      .
      .
      .

```

FIGURE 4.9 Base addressing.

(see below) at load time). If one wanted to access segment two, one would load its address value into the base register, LDADR BR,TWO, and perform similar computations:

```

LDA  C(BR) → 2000 + 0 → 2000
ADD  D(BR) → 2000 + 1 → 2001.

```

The command LDADR stands for load address, as opposed to LDA (load accumulator).

One can use segment addressing to separate the code (instructions) from the data in writing a program. For example, in Figure 4.11, the value of the PC has the address of the start address in the code segment, CS. This is computed by adding the value of the CS register (CS = 2000) to the relative starting address (zero in the program of Fig. 4.11): $2000 + 0 \rightarrow 2000$. The data are accessed by displacements from the data segment (DS) address, LDA A(DS), which indicates that the relative address of A, zero, is to be added to the value in our DS register, DS = 1000, to obtain the effective address. Using this method of segmenting data and instructions, it is more difficult to execute data "accidentally." This addressing scheme—code segments (CS) and data segments (DS)—is utilized on the IBM PC [3,23].

```

Segment Base:      1000
Address of A,
  relative to
  start of
  segment:         0
Effective Address: 1000

Segment Base:      1000
Address of B,
  relative to
  start of
  segment:         1
Effective Address: 1001

```

FIGURE 4.10 Calculations for effective address.

```
DS: 1000

A:5
B:6
C:RES 1

CS: 2000

START: LDA A(DS)
      ADD B(DS)
      STA C(DS)
      HLT

      Relative-START: 0 in CS
      PC: 2000 (2000 + 0)
```

FIGURE 4.11 Separating the code from the data.

4.3 RELATIVE AND ABSOLUTE ADDRESSES

Addresses such as the 0, 1, 2 of our assemblies are usually looked at as relative addresses, the location of the statements relative to the start of the program, counting from zero. This allows the program to be loaded into the main memory at locations other than that of location 0. This is the case in both machines that use base displacement addressing (IBM 360/370 series, IBM PC) and those that do not (UNIVAC 1108, DEC 10).

NOTE: Machines that do not use base displacement addressing often have some other mechanism for creating distinct segments. For example, the UNIVAC 1108 had 32 PCs that could be selected to control different segments. The purpose was the same: to separate data from instructions, as well as to increase machine performance by interleaving instruction fetches and data fetches.

The actual locations at which the program is loaded are called "absolute." The word absolute is intended to convey the fact that these are the actual (real) addresses on the machine (the hardware locations). The distinction is between the description of a house by its absolute address, 7117 S. Mozart, versus the description, "the second house on the left" (relative address).

The use of relative addresses in the assembler code allows one to load the program at any location in main memory. For example, if space is available in the main memory at location 4096, the program could be loaded there. The addresses of the program would then have to be adjusted to reflect the fact that the first instruction is no longer at address 0, but at address 4096. This is the job of the loader. The loader would take the relative addresses, 0, 1, and 2 and add 4096 to each of them to obtain the actual main memory addresses, the absolute addresses [21, 22].

The Loader

The loader is then a program that places the program being loaded (machine code version) into main memory. It does so by adjusting the addresses of the program to reflect the true load address, the physical start of the program, which will not always, if ever, be at location zero. We illustrate the process with a load address of 100, and the machine code version of the program of Figure 1.2, the code produced as Figure 2.10. In Figure 2.10, the program had relative addresses beginning at 0. Given the load location of 100, the loader would take the relative addresses, 0, 1, and 2 and add 100 to them to obtain the actual main memory addresses:

```
0: 5 → 0 + 100 → 100: 5
1: 6 → 1 + 100 → 101: 6
```

It would also have to adjust the addresses in the operand fields. To do so, it would need to know which fields contain addresses, as opposed to data values (the constants 5 and 6 in locations 0 and 1), which should not be adjusted. It distinguishes the relative addresses from the constants by having an assembler that marks the addresses with primes, or apostrophes (or their machine equivalent) as in Figure 4.12. The loaded program would look like that of Figure 4.13. All relative addresses, which were marked by the prime, have been changed to a value of 100 or greater (100 was the load address). The opcodes and data values have remained unchanged. (What would happen if the opcodes were adjusted? The data values? If the opcodes were adjusted, illegal instruction codes would be generated; if the data values were adjusted, wrong results would be obtained: $105 + 106 = 211$, not $5 + 6 = 11$, as intended.)

The Linker

The execution of the loader is usually preceded by the operation of the linker. The job of the linker is to make one module out of two or more modules. For example, if one has created the two program segments of Figure 4.14, one could link them into the single segment of Figure 4.15. The second segment is continued where the first left off, adjusting the "relocatable" address 0' to 7'. If the second segment had more locations, counting would continue from there: 8', 9', A', B', and so forth.

The actual operation of a linker is slightly more complex because it must resolve external references, references in one routine to locations in another routine (e.g., a call to a subroutine). However, the principle is the same:

```
0': 05
1': 06
2': 00
3': 40'
4': 21'
5': 52'
6': 60
```

START-PC: 3'

FIGURE 4.12 Distinguishing between relative addresses and constants.

```

100: 0 5
101: 0 6
102: 0 0
103: 4 100
104: 2 101
105: 5 102
106: 6 0

```

START-PC: 103

FIGURE 4.13 Program for Fig. 4.12.

The linker links segments; the loader then loads the final product (the "object module," which is a single module after the linker has done its work).

NOTE: The addresses are called relocatable because the loader can relocate them, adjusting their values to reflect the relocation. The constants in the program are not relocatable. Their value is absolute. Some addresses in an assembler program may also be absolute, such as the address of the AC in the command `MOV AC0,A`, which must be translated as `MOV 0,0'`. The zero on the left, the address of AC0 is absolute. It is the machine address of the AC, which sits in one physical place. The address on the right, `0'` (the address labeled A in the original program) is relocatable: A can be the name of any location in the main memory and does not have to be a specific location. The ACs, or the PC, are at specific locations, built into the hardware.

5.0 INSTRUCTION FORMATS

The last topic we cover is the format of instructions. We have assumed a single, simple format in the instructions for ENC. There are two fields, opcode and operand, each 4 bits wide, numbered from 0 through 7:

One

```

0': 0 5
1': 0 6
2': 0 0
3': 4 0'
4': 2 1'
5': 5 2'
6': 1 2'

```

START-PC: 3'

Two

```

0': 6 0

```

FIGURE 4.14 Two program segments.


```

0': 0 5
1': 0 6
2': 0 0
3': 4 0'
4': 2 1'
5': 5 2'
6': 1 2'
7': 6 0

```

START-PC: 3'

FIGURE 4.15 Single-segment program created using a linker.

Opcode	Operand
Bits: 7 6 5 4	3 2 1 0

The right-most 4 bits, bit 3 through bit 0, contain the operand field; the left-most 4 bits, bits 7 through 4, contain the opcode value.

Machines with more complex addressing schemes have more complex instruction formats. For example, if a machine has several ACs, bits will be set aside to indicate which accumulator is in use (Fig. 5.1). If indirect addressing is used, a bit will be set to indicate whether or not it is desired in a particular instruction (Fig. 5.2). If indexing is allowed, bits will be set aside to indicate the index register, if more than one index register is allowed (Fig. 5.3). A single bit (like the indirect bit), would be used if only one index register were available.

Hence, an instruction format might look like that of Figure 5.4. The 5 bits in the opcode would indicate that there can be up to 32 opcodes: $2^5 = 32$ (! = raised to the power of). The 6 bits in the memory field (5 through 0) indicate a total of 64 memory locations, which is rather small for a realistic computer.

Large computers solved the problem of having large instruction sets (requiring several opcode bits) and large memories (say, a million or so locations), requiring a large number of operand bits, by having large word sizes (e.g., 36 bits or 32 bits), as well as various forms of addressing (base displacement, indirect addressing, indexing, etc.). Microcomputers, with a more limited word size, such as 16 bits, solve the problem with various instruction formats, reflecting the various addressing schemes. These are explained in the reference manuals and texts for the particular machines.

AC Bits

x y

```

0 0 = AC 0
0 1 = AC 1
1 0 = AC 2
1 1 = AC 3

```

FIGURE 5.1 Instruction format for a complex machine.

Indirect Bit:

Indirect Bit = 0 (Not Desired)

Indirect Bit = 1 (Bit Lit, Desired)

FIGURE 5.2 Bit setting for indirect addressing.

The point is that each instruction set has a particular format. This is the format that indicates to the computer (by "soldering it in") how the bits in the IR (Fig. 1.1) are to be interpreted. The interpretation results in the execution of the instruction on the machine.

6.0 SUMMARY

We have treated the concepts of assembly language, assemblers, computer architecture (hardware and instruction set format), and linkers and loaders. Assembly language is the symbolic programming language associated with a particular machine. The assembler is a program that translates this symbolic language into the machine language of the machine. The linker and loader then prepare the modules resulting from the work of the assembler for execution. The execution takes place "in the hardware," by the electricity coursing through the electronic circuits and storage locations of main memory, for example, A, B, and C, and of the processing unit, for example, AC and PC, sometimes with interfaces to external (peripheral) devices, for example, the keyboard, the monitor, or secondary storage devices like the disk, floppy or hard, or tape drive. The entire process takes pages to explain and micro-seconds to happen (or seconds, depending on the machine and the instruction [operation] to be performed; if it takes longer, for example, as long as it takes to read an article about the process, the system is "sour.")

x y

0 0 = no indexing

0 1 = index by value in X1

1 0 = index by value in X2

1 1 = index by value in X3

FIGURE 5.3 Bits indicating index register.

Opcode	Register (AC)	Indirect?	Index Register	MEM
bits 15--11	bits 10--9	bit 8	bits 7--6	bits 5--0

FIGURE 5.4 Instruction format.

REFERENCES

1. Caxton C. Foster, *Computer Architecture*, 2nd ed., Van Nostrand Reinhold, New York, 1976.
2. Anthony Ralston, *Introduction to Programming and Computer Science*, McGraw-Hill, New York, 1971.
3. Robert Lafore, *Assembly Language Primer for the IBM PC and XT*, Plume/Waite Publishers, New York, 1984.
4. Peter Norton, *The Peter Norton Programmer's Guide to the IBM PC*, Microsoft Press, Bellevue, WA, 1985.
5. John J. Donovan, *Systems Programming*, McGraw-Hill, New York, 1972.
6. Sara Baase, *VAX-11 Assembly Language Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1983.
7. Nancy Stern, *From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers*, Digital Press, Bedford, MA, 1981.
8. Herman H. Goldstine, *The Computer from Pascal to von Neumann*, Princeton University Press, Princeton, NJ, 1972.
9. Simon Lavington, *Early British Computers*, Digital Press, Bedford, MA, 1980.
10. Kent C. Redmond and Thomas M. Smith, *Project Whirlwind: The History of a Pioneer Computer*, Digital Press, Bedford, MA, 1980.
11. Harold S. Stone, ed., *Introduction to Computer Architecture*, Science Research Associates, Chicago, IL, 1975.
12. M. E. Sloan, *Computer Hardware and Organization*, Science Research Associates, Chicago, IL, 1983.
13. John L. Hughes, *Digital Computer Lab Workbook*, Digital Press, Bedford, MA, 1968.
14. Jean-Loup Baer, *Computer Systems Architecture*, Computer Science Press, Rockville, MD, 1980.
15. V. Carl Hamacher, Zvonko G. Vranesic, and Safwat G. Zaky, *Computer Organization*, 2nd ed., McGraw-Hill, New York, 1984.
16. Ivan Tomek, *Introduction to Computer Organization*, Computer Science Press, Rockville, MD, 1981.
17. M. Morris Mano, *Digital Design*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
18. Gerhard E. Hoernes and Melvin F. Heilweil, *Introduction to Boolean Algebra and Logic Design: A Program for Self-Instruction*, McGraw-Hill, New York, 1964.
19. Henry M. Levy and Richard H. Eckhouse, Jr., *Computer Programming and Architecture—The VAX-11*, Digital Press, Bedford, MA, 1980.
20. C. William Gear, *Computer Organization and Programming*, McGraw-Hill, New York, 1969.
21. D. W. Barron, *Assembler and Loaders*, 3rd ed., Elsevier North-Holland, New York, 1978.
22. Robert M. Graham, *Principles of Systems Programming*, John Wiley, New York, 1975.
23. Alan R. Miller, *Assembly Language Techniques for the IBM PC*, SYBEX, Berkeley, CA, 1986.
24. John F. Wakerly, *Microcomputer Architecture and Programming*, John Wiley, New York, 1981.
25. S. E. Greenfield, *The Architecture of Microcomputers*, Winthrop Publishers, Cambridge, MA, 1980.

26. Claude A. Wiatrowski and Charles H. House, *Logic Circuits and Microcomputer Systems*, McGraw-Hill, New York, 1980.
27. Gene A. Streitmatter and Vito Fiore, *Microprocessors Theory and Applications*, Reston Publishing, Reston, VA, 1982.
28. Nancy Stern, Alden Sager, and Robert A. Stern, *Assembler Language Programming for IBM and IBM-Compatible Computers*, John Wiley, New York, 1986.
29. Arthur Gill, *Machine and Assembly Language Programming of the PDP-11*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
30. Glenn H. MacEwen, *Introduction to Computer Systems: Using the PDP-11 and Pascal*, McGraw-Hill, New York, 1980.

ROGER FLYNN

